



TECHNISCHE
UNIVERSITÄT
DARMSTADT

ULB

Automatic Identification and Recovery of Obfuscated Android Apps

Glanz, Leonid
(2020)

DOI (TUprints): <https://doi.org/10.25534/tuprints-00014647>

License:



CC-BY-SA 4.0 International - Creative Commons, Attribution Share-alike

Publication type: Ph.D. Thesis

Division: 20 Department of Computer Science

Original source: <https://tuprints.ulb.tu-darmstadt.de/14647>



Automatic Identification and Recovery of Obfuscated Android Apps

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Leonid Glanz, M.Sc.

geboren in Karabulak (Kasachstan).

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr.-Ing. Awais Rashid
Datum der Einreichung:	15. Oktober 2020
Datum der mündlichen Prüfung:	25. November 2020

Erscheinungsjahr 2020

Darmstädter Dissertationen

D17

Glanz, Leonid : Automatic Identification and Recovery of Obfuscated Android Apps
Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUPrints: 2020

URN: urn:nbn:de:tuda-tuprints-146479

URL: <https://tuprints.ulb.tu-darmstadt.de/id/eprint/14647>

Tag der mündlichen Prüfung: 25.11.2020

Veröffentlicht unter CC BY-SA 4.0 International
<https://creativecommons.org/licenses/>

Preface

While looking for a method to break big problems into small solvable units, I discovered programming for me. As a bachelor student, I became aware of the Goal-Question-Metric approach (GQM). It allowed me to make all goals accessible by asking detailed questions and using appropriate measurement methods to answer them.

In the course of my master's studies, I became aware of code analysis and obfuscation, which offered me an exciting challenge and demanded my knowledge. This led to my decision to pursue these two topics in my dissertation.

The GQM approach allowed me to convert all goals into measurement parameters. However, I could not always define the exact thresholds for the measurements from which action should be taken. Since these thresholds often depended on unknown criteria, I taught myself to use machine learning techniques during my time as a Ph.D. student. With these techniques, I was able to extract the desired thresholds as patterns and rules from previous decisions and the collected data.

Using all these techniques, I have designed and developed, in this dissertation, a collection of deobfuscation approaches. These approaches support analysts in the automatic identification and recovering of obfuscated Android apps. I started my Ph.D. student job in the project of Ben Herman under the supervision of Professor Mira Mezini in the Software Technology Group (STG) at the Technische Universität Darmstadt. In this project, I learned more about vulnerability and malware detection, which helped me to find an application for my deobfuscation challenge. Afterward, I worked on other projects for several years, which deepened my knowledge and led to this dissertation.

My life as well as this dissertation would have been more difficult without the help of others. In the following, I try to thank all the people who supported me on my way:

First of all, I would like to thank my supervisor Professor Mira Mezini, who showed me new ways that demanded and enhanced my knowledge of software engineering and IT security. She helped me develop a clearer view of these topics and improve the structure of my thinking processes. Thank you for all your efforts that supported me to get to this point of the long doctoral process.

I also want to thank Professor Awais Rashid for being the second examiner of my dissertation. I am grateful for the long hours you spent on carefully reviewing my work.

My work would not have gotten this far without the help of people from the security subgroup with whom I could discuss my ideas. I would like to thank Lars Baumgärtner, Michael Eichberg, Dominik Helm, Ben Hermann, Florian Kübler, Johannes Lerch, Patrick Müller, Krishna Narasimhan, Michael Reif, and Anna-Katharina Wickert for your feedback that gave me new thinking impulses and for your patience when I presented new ideas involving black magic (machine learning).

Over the years, I enjoyed working with great students who helped me advance my

research projects and did a great deal of the implementation work. I would like to thank Florian Breithfelder, Mattis Manfred Kämmerer, Patrick Müller, and Jonas Schlitzer and hope that they continue to be successful in their careers.

Of course, I also want to thank my colleagues, who took the time to proofread this thesis and gave me their feedback. First of all, I would like to thank Lars Baumgärtner, who read several chapters over and over again. Then I would like to thank Sven Amann, Dominik Helm, Ragnar Mogk, Patrick Müller, Krishna Narasimhan, Michael Reif, Aditya Oak, and Pascal Weisenburger. Their feedback helped me to improve the clarity of this thesis.

The last years would not be as enjoyable and fruitful without my colleagues Sven Amann, Lars Baumgärtner, Andi Bejleri, Oliver Bracevac, Ervina Cergani, Joscha Drechsler, Michael Eichberg, Matthias Eichholz, Sebastian Erdweg, Sylvia Grewe, Dominik Helm, Ben Hermann, Sven Keidel, Matthias Krebs, Mirko Köhler, Florian Kübler, Edlira Kuci, Johannes Lerch, Ingo Maier, Nafise Eskandani Masoule, Annette Miller, Ragnar Mogk, Patrick Müller, Sarah Nadi, Krishna Narasimhan, Aditya Oak, Sebastian Proksch, Michael Reif, David Richter, Tobias Roth, Guido Salvaneschi, Simon Schönwälder, Jan Sinschek, Daniel Sokolowski, Jurgen van Ham, Manuel Weiel, Pascal Weisenburger, and Anna-Katharina Wickert.

Ultimately, I would like to thank Gudrun Harris and Claudia Roßman. Gudrun, you are the soul of the STG and a master of administration. I would surely have fallen into some pitfalls when it concerns paperwork. You ensured that we were well funded, fulfilled all regulations, and always had an open ear for our troubles. Claudia Roßman will be your successor, and I want to thank her for the help with the dissertation registration. I hope we will work together as well as we did with Gudrun. Many thanks to you both for the continued assistance.

Last but not least, I would like to thank my wife, my family, and my friends. Thank you all for the support with small and bigger steps in my life, and I want to share many more amazing moments with you.

Editorial notice: Throughout this thesis, I use the term “we” and “us” to describe my work. This is meant to underline that research is always a cooperative effort. I would have much less (if something at all) to present here if other people had not taken time off their own work to review, discuss, and contribute to mine. I am deeply grateful for their effort.

Abstract

Every day, developers add new applications (apps) to the Google Play Store, which ease users' lives and entertain them. The rapid development of these apps is only possible through the provision of software libraries whose functionality can be directly integrated into an app without creating it from scratch. For instance, some libraries provide extended possibilities for displaying content or additional support for specific networking capabilities. All these libraries are bundled with the main application code into one binary to avoid delays due to the loading of external functionality.

The wide distribution of Android apps and the high turnover in this market attracts criminal actors (attackers). These attackers decompile the apps, integrate additional ad libraries or malware, and republish them. Through this approach, they use the apps' popularity to trick users into downloading their repackaged apps. To prevent such malicious practices, developers and producers of libraries have begun to obfuscate their apps to make the decompilation process more challenging. However, the obfuscation of apps is not only done by developers but also by attackers to make the detection of copyright infringement harder or hide their malicious intent.

While analysts try to protect the developers' copyright and the privacy and security of app users, code obfuscation hinders them from identifying libraries and repackaged apps, detecting obfuscated names and strings, and recovering them. The obfuscated code might contain not only malware but also vulnerabilities or unauthorized access to private data.

This dissertation introduces different approaches that support the analyses mentioned above using static analysis, dynamic analysis, and machine learning. Since the obfuscation of repackaged apps makes it difficult to distinguish between library and app code, we present approaches for library detection, separation of app code, and mapping of library code. We evaluated the effectiveness of these approaches under the influence of different obfuscation techniques. Furthermore, we present our approach for identifying repackaged apps that uses our library identification to measure the similarity between repackaged and original apps without the influence of library code, which would distort the measurement. Further contributions support the recovery of names from obfuscated entities in library code. Finally, we presented an approach for identifying and recovering obfuscated strings that supports data-flow analyses.

Using our approaches, we outperformed all state-of-the-art competitors. Furthermore, we analyzed in total over 100,000 apps for obfuscated names, obfuscated libraries, obfuscated strings, and repackaged apps.

Zusammenfassung

Jeden Tag fügen Entwickler dem Google Play Store neue Anwendungen (Apps) hinzu, die das Leben der Nutzer erleichtern und sie unterhalten. Die rapide Entwicklung dieser Apps ist nur durch die Bereitstellung von Softwarebibliotheken möglich, deren Funktionalität direkt in eine App integriert werden kann, ohne diese von Grund auf neu zu erstellen. So bieten einige Bibliotheken beispielsweise erweiterte Möglichkeiten zur Anzeige von Inhalten oder zusätzliche Unterstützung für bestimmte Netzwerkfähigkeiten. Die Bibliotheken werden mit dem Hauptanwendungscode in einer Binärdatei gebündelt, um Verzögerungen durch das Laden externer Funktionalität zu vermeiden.

Die weite Verbreitung von Android-Apps und der hohe Umsatz in diesem Markt zieht kriminelle Akteure (Angreifer) an. Diese Angreifer dekompile die Apps, integrieren zusätzliche Werbibibliotheken oder Malware und veröffentlichen sie erneut. Auf diese Weise nutzen sie die Popularität der Apps, um Benutzer zu verleiten, ihre neu verpackten Apps herunterzuladen. Um solche bösartigen Praktiken zu verhindern, haben Entwickler und Hersteller von Bibliotheken damit begonnen, ihre Anwendungen zu verschleiern, um den Dekompilierungsprozess schwieriger zu gestalten. Die Verschleierung von Anwendungen wird jedoch nicht nur von Entwicklern, sondern auch von Angreifern vorgenommen, um die Aufdeckung von Urheberrechtsverletzungen zu erschweren oder ihre böswilligen Absichten zu verbergen.

Während Analysten versuchen, das Urheberrecht der Entwickler, die Privatsphäre und Sicherheit der Benutzer von Anwendungen zu schützen, hindert die Code-Verschleierung sie daran, Bibliotheken und neu-verpackte Apps zu identifizieren, verschleierte Namen und Zeichenketten zu erkennen und wiederherzustellen. Der verschleierte Code kann nicht nur Malware, sondern auch Schwachstellen oder unbefugten Zugriff auf private Daten beinhalten.

In dieser Dissertation werden verschiedene Ansätze vorgestellt, die die oben genannten Analysen mit Hilfe von statischer Analyse, dynamischer Analyse und maschinellem Lernen unterstützen. Da die Verschleierung von neu-verpackten Apps es schwierig macht, zwischen Bibliotheks- und Anwendungscode zu unterscheiden, stellen wir Ansätze zur Erkennung von Bibliotheken, zur Trennung von Anwendungscode und zum Wiederherstellen von Namen von Bibliothekscodeentitäten vor. Wir haben die Wirksamkeit dieser Ansätze unter dem Einfluss verschiedener Verschleierungstechniken analysiert. Des Weiteren stellen wir unseren Ansatz zur Identifizierung von neu-verpackten Apps vor, der unsere Bibliotheksidentifikation zur Messung der Ähnlichkeit zwischen um-verpackten und Originalanwendungen ohne den Einfluss von Bibliothekscode verwendet. Weitere Beiträge unterstützen die Wiederherstellung von Namen aus verschleierten Entitäten im Bibliothekscode. Schließlich stellten wir einen Ansatz zur Identifizierung und Wiederherstellung von verschleierten Zeichenketten vor, der Datenflussanalysen unterstützt.

Mit unseren Ansätzen übertreffen wir alle aktuellen Konkurrenten und haben insgesamt über 100.000 Anwendungen auf verschleierte Namen, verschleierte Bibliotheken, verschleierte Zeichenfolgen und neu-verpackte Apps analysiert.

Contents

Preface	3
1. Introduction	15
1.1. Problem Statement	15
1.2. Contributions of this Thesis	17
1.3. Structure of this Thesis	22
1.4. Publications	23
2. Background	25
2.1. Android Application Architecture	25
2.1.1. Android Manifest	26
2.1.2. Executable Code	28
2.2. Obfuscation Techniques	29
2.2.1. Obfuscation of Android Apps	29
2.3. Machine Learning	32
2.3.1. Clustering	32
2.3.2. Classification	34
I. Obfuscated Libraries	39
Obfuscated Library Detection and Separation of App Code	41
3. Library Detection	43
3.1. State-of-the-Art Library Detectors	44
3.2. LibDetect	48
3.2.1. Representation	49
3.2.2. Lookup	52
3.2.3. Method/Class Matcher	53
3.3. Evaluation	54
3.3.1. Robustness of Code Representation	54
3.3.2. Library Detection	57
3.3.3. Summary	58
3.4. Threats to Validity	59
3.5. Conclusion	59
4. App-Code Separation	61
4.1. State-of-the-Art App-Code Separators	62

4.2.	AppSeparator	64
4.2.1.	Extraction	64
4.2.2.	Classification	66
4.3.	Evaluation	67
4.3.1.	Dataset	68
4.3.2.	Classifier Selection for AppSeparator	68
4.3.3.	Comparison against other Library Detectors	69
4.3.4.	Performance on Obfuscated Apps	70
4.3.5.	Summary	72
4.4.	Threats to Validity	72
4.5.	Conclusion	73
5.	Repackage Detection	75
5.1.	Attacker Model	78
5.2.	State of the Art	78
5.3.	CodeMatch	82
5.4.	Evaluation	84
5.4.1.	Comparison Against Other Tools	84
5.4.2.	App Data-Provision & Insights	85
5.5.	Threats to Validity	86
5.6.	Conclusion	87
	Summary	89
II.	Obfuscated Names	91
	Obfuscated Name Detection and Recovery	93
6.	Library Mapping	95
6.1.	State-of-the-Art of Library Mapping Approaches	96
6.2.	Case Study of Collision Rate from Method Representations	98
6.2.1.	Data Set	99
6.2.2.	Results	99
6.3.	LibMapper	101
6.3.1.	AppSeparator	102
6.3.2.	Match (Structurally) Unchanged Classes	102
6.3.3.	Match Classes by Methods	103
6.3.4.	Merge Classes	104
6.3.5.	Match Method Data	104
6.3.6.	Match Field Data	107
6.3.7.	Match Class Data	109
6.3.8.	Preferred Classes	111
6.3.9.	Resolve Mapping	111

6.4.	Evaluation	111
6.4.1.	Standard Name Obfuscation Configuration	112
6.4.2.	Advanced Name Obfuscation Configuration	112
6.4.3.	False Negatives of LibMapper	114
6.4.4.	Discussion	115
6.5.	Threats to Validity	116
6.6.	Conclusion	116
7.	Detection of Obfuscated Names	119
7.1.	State-of-the-Art Detectors of Obfuscated Names	120
7.2.	ObfusSpot	122
7.2.1.	Anomaly Classification	123
7.2.2.	Frequency Classification	125
7.2.3.	Library Mapping	126
7.2.4.	Name Matcher	126
7.3.	Evaluation	127
7.3.1.	Selection of Classifiers	127
7.3.2.	Effectiveness of <i>ObfusSpot</i>	129
7.3.3.	Name Obfuscation in the Wild	132
7.4.	Threats to Validity	134
7.5.	Conclusion	134
	Summary	137
III.	Obfuscated Strings	139
	Obfuscated String Detection and Recovering	141
8.	Study of String Obfuscation Techniques	143
8.1.	Dataset	143
8.2.	Methodology	144
8.3.	Overview of Identified Techniques	145
8.4.	Identified Concepts of String Obfuscation	148
8.5.	Threats to Validity	151
8.6.	Conclusion	151
9.	Detection of Obfuscated Strings	153
9.1.	State-of-the-Art Detectors of String Obfuscation	153
9.2.	String Classifier	155
9.3.	Evaluation	159
9.3.1.	Classifier Comparison	159
9.3.2.	Effectiveness of the String Classifier	160
9.3.3.	Falsely Classified Strings	161
9.4.	Method Classifier	162

9.5. Evaluation	165
9.5.1. Identification of Deobfuscation-Method Variations	165
9.5.2. Comparison With Other Detectors	167
9.5.3. Falsely Classified Deobfuscation Methods	169
9.5.4. Discussion	170
9.6. Threats to Validity	171
9.7. Conclusion	171
10.String Deobfuscation	173
10.1. State-of-the-Art String Deobfuscators	174
10.2. StringHound	179
10.2.1. Slicing Relevant String Usages	179
10.2.2. Our targeted Slicing	180
10.2.3. Executing Sliced String Usages	184
10.3. Evaluation	186
10.3.1. Comparison with Other Deobfuscators	187
10.3.2. Findings in the Wild	188
10.3.3. Runtime Performance	194
10.4. Threats to Validity	196
10.5. Conclusion	196
Summary	197
IV. Conclusion and Outlook	199
11.Conclusion	201
11.1. Summary of Results	202
11.1.1. Library Detection	202
11.1.2. App Code Separation	203
11.1.3. Repackaging Detection	203
11.1.4. Library Mapping	204
11.1.5. Obfuscated Name Detection	204
11.1.6. Obfuscated String Detection	205
11.1.7. String Deobfuscation	205
11.2. Closing Discussion	206
12.Future Work	209
12.1. Increasing Processing Efficiency	209
12.2. Keeping Representations Up-to-Date	209
12.3. Handling Underlying Structural Changes	210
12.4. Handling the Hiding of Functionality	211
12.5. Eliminating the Arms Race	211

Contributed Implementations and Data	213
12.6. CodeMatch & LibDetect	213
12.7. StringHound, String Classifier & Method Classifier	213
12.8. AppSeparator, LibMapper, & ObfusSpot	213
 Bibliography	 215
 Appendix	 237

1. Introduction

Google launched the Play Store in 2008 [And20d], and since then, the store contains over 2,960,000 apps [num20]. With this large number of apps, users can find their favorite app for gaming, social networks, finance, sports, or other activities. Currently, it is even possible for users to develop specific apps for themselves with only restricted knowledge of software engineering using app generators [ODS⁺18]. While the functionality of app generators is limited, software libraries allow developers to integrate new functionality into their apps without writing it from scratch. Their integration boosts the development speed of new apps and might be crucial for the Google Play store’s growth [gro20].

The large userbase of Android apps [Ann20] attracts malicious actors. They use the reputation of established apps to trick users into downloading their payload. For that, the malicious actors decompile apps and republish them to get financial gains from inserting additional ad libraries or accessing sensitive user data (e.g., credit card numbers, bank access data) by using integrated malware [STDA⁺17, ARSC16, AMSS15, FGL⁺13, FLB⁺15, LLCT13]. This practice may harm the developer’s reputation of established apps and contribute to user distrust in the Android ecosystem. The Cybercrime Magazine [dam20] estimates that by 2021 the annual loss through cybercrime, including repackaging and other attacks, will amount to six trillion dollars.

To nurture a healthy Android ecosystem, analysts investigate the content of libraries and apps that can pose a threat to a user’s privacy and security. During such an inspection, analysts use automated approaches [YFM⁺17, IWAM17, STDA⁺17, and17, LLB⁺17] to detect repackaged apps and inform potentially affected developers. Furthermore, analysts identify sensitive data that flows to untrusted destinations to warn developers and users of potential dangers [ARF⁺14, RAMB16, ZWWJ15, BP18, WL18, MW17, CYL⁺17, CFL⁺17]. In their investigations, analysts are hindered by obfuscated code and data. Another difficulty for analysts is the compilation of app and library code into a single binary since, after obfuscation, the code cannot be distinguished from one another and needs to be analyzed as well, although the library code is known and often not dangerous. As analysts address these challenges, they must identify and recover obfuscated data and code locations. Since they cannot perform these tasks for millions of apps manually, they need automated approaches. This thesis makes important contributions for automating the identification and recovery process of libraries, names of code entities, and strings in obfuscated apps.

1.1. Problem Statement

In recent years, many approaches [LWW⁺17, MWGC16, BBD16, ZDZ⁺18, WWZR18, FR19, ZBK19] analyzed apps to identify libraries or separate them from the app code-

1. Introduction

base. The application scenarios of such approaches varied from the detection of vulnerabilities in libraries, license violations, or the separation of app code to find repackaged apps. The approaches are mainly hindered by obfuscated code, and that app and library code is compiled into a single binary file, which makes library code indistinguishable from app code. Additionally, most library detectors use very restricted representations that are not sufficient to detect obfuscated libraries. For example, some detectors use white lists of known library package names to detect libraries. If all package names in an app were obfuscated, the white lists are useless. Furthermore, library detectors that are used to separate app code from library code often maintain a database of all methods to identify the exact version of a library. However, such a database can never be complete because there is no central source that could be checked for completeness.

Due to library detectors' mentioned limitations, other approaches using these detectors can only perform their functions to a limited extent. For example, analysts have developed more than 40 different repackaging detection approaches [LBK19] that inspect data flows, control flows, and views to extract heuristics for their static and dynamic analyses. However, all these approaches only insufficiently identify libraries in potentially repackaged apps or are vulnerable to various obfuscation techniques. If a repackaging detection approach misses filtering out library code, it could lead to false positives and false negatives. On the one hand, false positives could occur because more than 60% of the sub-packages in an app belong to library code [WGMC15]. This high percentage of library code increases the similarity measurements used by repackaging detection approaches and let apps that use the same libraries seem to be repackaged. On the other hand, false negatives could be caused by malicious actors who insert additional code into repackaged apps to reduce the similarity values of the compared apps.

Repackaging detectors profit from improved library detection but also all approaches that analyze the entire codebase to examine the data flow [ARF⁺14, RAMB16, ZWWJ15, BP18, WL18, MW17, CYL⁺17, CFL⁺17] or recover obfuscated code [BRTV16, RVK15, VCD17, Jaf17, AZLY19, CFPK20]. For instance, if an analysis identifies that a library contains untrusted data flows, it directly can flag all apps that use this library as containing the same untrusted flow. Therefore, using a library detector improves the speed of app analyses and enables the analyses to deal with large numbers of apps.

The detection of library code supports many approaches. However, these approaches are still challenged by obfuscation techniques. For instance, some of the approaches struggle with repackaged apps that contain restructured and renamed code that may lead to a lower similarity between the repackaged and the original app. Furthermore, if analysts ignore obfuscated code or data during their investigations, they might fail to identify sensitive information and come to undesired conclusions.

Code and data obfuscation challenges not only repackaging and library detectors but also approaches that try to infer names for obfuscated code entities, such as classes, fields, and methods. We refer to these approaches as name inference approaches [BRTV16, RVK15, VCD17, Jaf17, AZLY19, CFPK20]. They use large numbers of repositories that contain different projects to learn a mapping between the context and the name of a code entity. However, these projects may also contain obfuscated code entities. Even large open-source repositories such as Maven Central [Mav17] or GitHub [git20]

contain obfuscated code entities. Additionally, while current approaches [WHA⁺18, Mir18, WR17, Don18] can easily detect manipulated names of code entities produced by standard obfuscators, some may circumvent detection using more advanced name manipulations. Furthermore, most of the approaches that detect obfuscated names can only identify whether the content of an app contains obfuscated names instead of locating each individual obfuscated name. Similar limitations have approaches that try to recover obfuscated strings [Dex20, sim20, Jav20a, Dex19b, RAMB16, BP18, WL18, ZWWJ15]. Instead of identifying the exact obfuscated string, the approaches analyze all strings that flow into previously specified methods. The tracking and analysis of all flows of these strings generate a huge time overhead that inhibits these approaches to process the large number of apps mentioned earlier. An additional limitation of some of these approaches is that they focus on a limited number of known string obfuscation techniques. Rather than exploiting the integrated code that is used by all obfuscators to deobfuscate the strings at the needed positions [WHA⁺18].

This dissertation contributes multiple approaches to overcome the drawbacks of others and advance research of the above-described topics. For this purpose, the dissertation focuses on:

- the identification of library names, versions, and classes,
- the separation of app code,
- the detection of repackaged apps,
- the detection of obfuscated class, field and method names,
- the recovery of names of library code entities,
- the identification of obfuscated strings,
- and their recovery.

1.2. Contributions of this Thesis

Figure 1.1 gives an overview of the abstract goal of this thesis, its challenges, and the contributions (lower left-hand side). The abstract challenges are derived from general definitions of obfuscation techniques [CTL97], and the contributions involve steps security analysts perform to examine unknown code [gui17, BRTV16, RAMB16]. First, they identify known code parts using library identification and repackaging detection. Second, they recover these parts using code deobfuscation. Finally, they examine the revealed data after deobfuscating it.

From the abstract items on the left-hand side, we present the items on the right-hand side, starting at the top with the analyses that are affected by our contributions, continuing with the concrete challenges that arose during the pursuit of our goal, and ending at the bottom with the subject areas to which we contribute our approaches.

While Figure 1.1 presents the different topics of this thesis, Figure 1.2 shows the relation between the different topics. Library detection and app code separation are used by

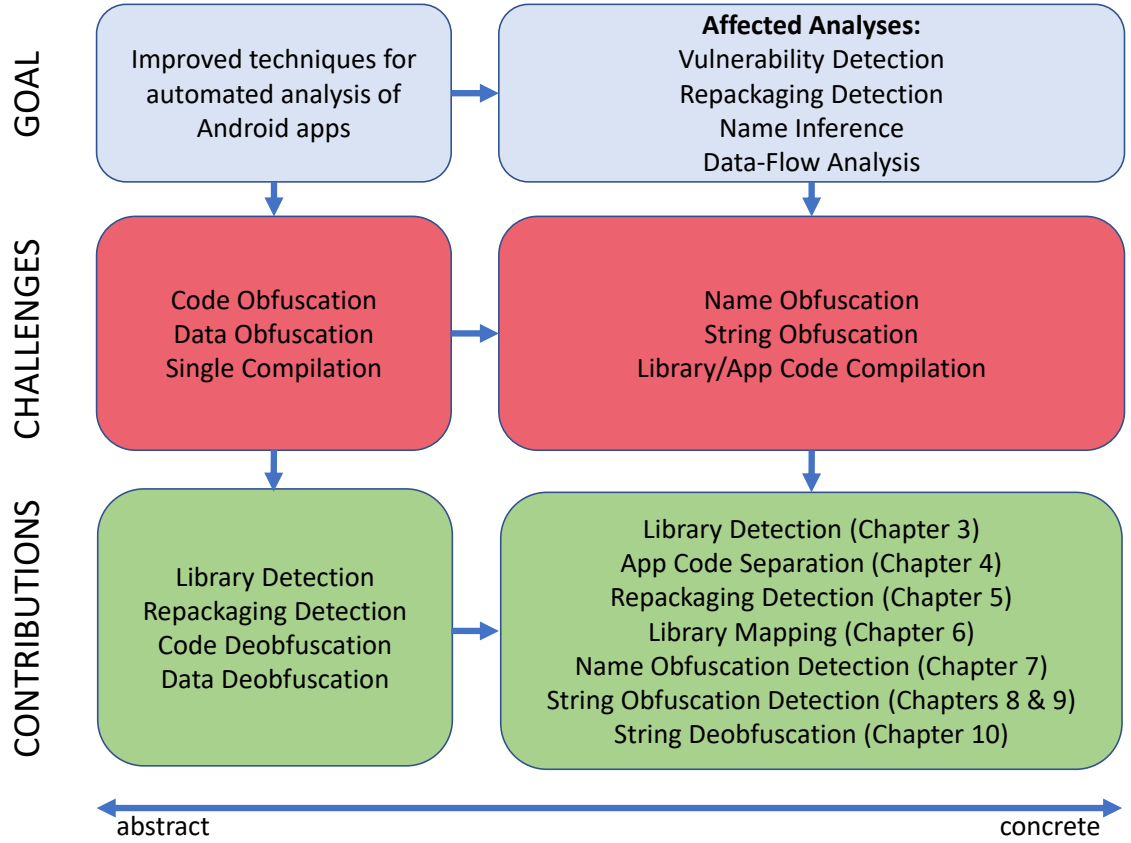


Figure 1.1.: Overview of dissertation

the repackaging detection to filter out libraries and thus reduce false positives. The two topics also support the mapping of original code-entity names to their obfuscated names for libraries. In turn, the library mapping contributes to the detection of obfuscated names, since manipulated library names can often provide insights into the patterns used to obfuscate other code entities. Finally, the detection of obfuscated strings is used in string deobfuscation. An approach that tries to reveal obfuscated strings without detecting the exact location of obfuscated strings is forced to try all possible strings in an app, causing a tremendous effort to analyze millions of apps.

The following paragraphs briefly summarize the concrete contributions of this thesis depicted in Figure 1.2.

1. Library Detection The first contribution is an approach to identify the name, version, and all classes of a library. Analysts can use this approach to detect vulnerable versions of library code or filter out library code for repackaging detection. We developed five code representations with different precision/recall trade-offs between handling more complex obfuscation techniques and identifying a code piece as precise as possible. Using these five representations, our tool *LibDetect* identifies library methods and uses the

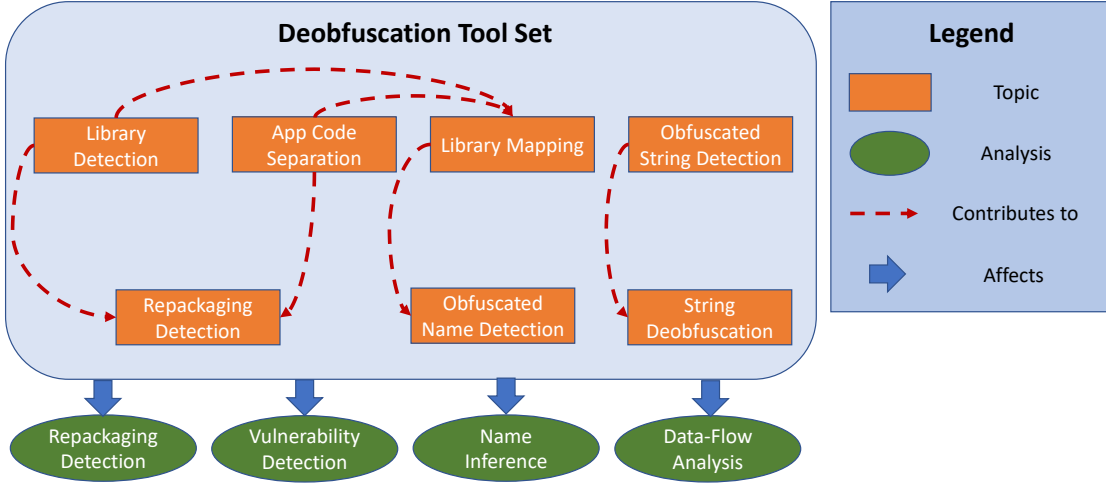


Figure 1.2.: Overview of relations between contributions

methods to deduce classes, packages, and the version of a library. We evaluated *LibDetect* with state-of-the-art library detection approaches on a ground truth of 1,000 apps, and it outperformed the other approaches by orders of magnitude. To facilitate further research, we provide our approach and the 1,000 apps that we used to evaluate all library detection approaches in a public repository.

2. App Code Separation An approach that identifies vulnerable versions of a library requires detailed information about each class of this library. However, if this approach is used to separate app code from library code, this information slows down the processing speed of the approach. For instance, repackaging detection needs only the code of an app and not the library code for its comparison algorithm. In the second contribution, we developed an app-code separation approach using static analysis and machine learning to support repackaging detectors. Our tool *AppSeparator* does not need to store the library’s entire code representation to split library code from app code. It is based on the assumption that app code accesses library code but not vice versa because library code was compiled as a single module before its integration into an app [AL12].

With this assumption, *AppSeparator* extracts all entry points of an app and analyzes which classes access these entry points transitively. Since obfuscation and optimization can blur the lines between library code and app code, *AppSeparator* additionally extracts the incoming and outgoing accesses to all classes and use a classifier to predict whether a class belongs to app or library code.

Given our app-code separation approach, an analyst can use it for repackaging detection and other measurements for which the analyst requires only the app code. During our experiments, *AppSeparator* outperformed all state-of-the-art approaches that use library detection to separate app code.

1. Introduction

3. Repackaging Detection The third contribution is our approach *CodeMatch* that detects repackaged apps in three steps: First, it filters out the library code of an app. Second, it uses an abstract representation of the remaining app code and uses fuzzy hashing to transform the representation into a fingerprint. Finally, *CodeMatch* uses this fingerprint to compare repackaged apps with the fingerprints of original apps. Using our approach can assist analysts in identifying repackaged apps that are commonly used to spread malicious content [LBK19].

We evaluated *CodeMatch* with four state-of-the-art repackaging detectors and *CodeMatch* outperformed them all. In the end, we measured the impact of repackaged apps on five different app stores. For this analysis, we downloaded more than 47,000 apps and examined them using *CodeMatch*. The analysis showed that about 15% of the apps are repackaged. We do not only provide the implementation of our approach but also a data set of 1,000 manually inspected apps that can be used to evaluate repackaging detectors.

4. Library Mapping While the first contribution only identifies which classes belong to which library version, the fourth contribution analyzes the relation between library code entities and their names, such as class names, field names, and method names. Using the first and second contribution, we developed *LibMapper* that identifies obfuscated names of library code entities and maps to them their original names from non-obfuscated library code. First, *LibMapper* uses *AppSeparator* and *LibDetect* to identify all libraries. Afterward, it uses *LibDetect* to identify all potential fully-qualified class names that might be a non-obfuscated counterpart for the class under analysis. Using these class names, *LibMapper* determines appropriate mappings for the field names and the method names. In the end, *LibMapper* uses the most suitable mapping to assign it to the code entities. We refer to the process of mapping original names of library code entities to their obfuscated counterparts as library mapping.

We compared *LibMapper* with the state-of-the-art library mapping approach for Android apps [BRTV16], and *LibMapper* outperformed the approach in all our experimental settings. Analysts can use *LibMapper* to understand obfuscated apps and examine which parts of the library code is used by malware.

5. Obfuscated Name Detection The fifth contribution introduces *ObfusSpot* an approach that identifies obfuscated names of code entities. First, it uses anomaly detection to identify obfuscated names that differ from non-obfuscated ones by their characteristics. Second, *ObfusSpot* analyzes an app for names that occur exceptionally frequent because non-obfuscated apps contain, on average, more unique names. In the last step, *ObfusSpot* uses library mapping to identify naming patterns of obfuscated library code entities and use these patterns to identify obfuscated names of app code entities with the same patterns.

Using *ObfusSpot*, we encountered code entities with obfuscated names in 99.99% of the 100,000 investigated apps. However, only 29% of the apps contained obfuscated names in their main code, and the rest of the obfuscated names were contained in the library code.

Furthermore, we found that some libraries are only released in obfuscated form. Analysts can benefit from using *ObfusSpot* in two forms: First, it can be used to prevent obfuscated names from being included in the learning process of approaches that infer names for code entities. Second, these approaches can also use *ObfusSpot* to identify obfuscated names and selectively replace them. In contrast to other detectors of obfuscated names, *ObfusSpot* can identify the position of an obfuscated name, instead of only determining that an app contains obfuscated names [WHA⁺18, Mir18, WR17, Don18].

6. Obfuscated String Detection In the sixth contribution, we conducted a study of string obfuscation techniques in ad libraries of 100,000 randomly selected apps. In this study, we selected all unique strings and manually analyzed whether they were manipulated using obfuscation. Furthermore, we singled out all unique method calls that return a string to manually analyze whether they are used to process obfuscated strings that are hidden in other data structures. Using this procedure, we identified 21 different string obfuscation techniques that protect the content of the strings from unauthorized extraction.

After examining these techniques, we created two approaches that complement each other and use machine learning. The first one classifies obfuscated strings based on their characteristics, and the second one identifies the logic used to deobfuscate a string [WHA⁺18]. Using our detectors can assist other approaches in identifying obfuscated strings. These approaches do not need to examine all strings in an app but can focus on the obfuscated ones to analyze their content or deobfuscate them.

For the evaluation of our detectors, we obfuscated the strings of apps that were previously not obfuscated using the 21 techniques found in our study. Afterward, we executed the detectors to determine whether they identify the obfuscated strings or the used de-obfuscation code. The results show that both detectors are very powerful and can detect all obfuscated strings.

7. String Deobfuscation In the last contribution, we developed *StringHound* an approach that uses the detectors from the previous contribution to identify obfuscated string to deobfuscate them. For the deobfuscation, *StringHound* uses slicing [Wei81] to extract the deobfuscation logic and all necessary values of an obfuscated string to reveal its content. By extracting as much context of the obfuscated strings as possible, *StringHound* circumvents anti-deobfuscation techniques.

We evaluated *StringHound* against four other string-deobfuscators using a data set that we obfuscated using the techniques identified in the study of string obfuscation techniques. *StringHound* outperformed the other tools by orders of magnitude. Furthermore, using *StringHound*, we analyzed 100,000 apps and identified vulnerabilities and accesses to private data hidden behind obfuscated strings. Moreover, our approach revealed obfuscated strings in malware and apps installed on over 100 million devices. During this analysis, we noticed that obfuscated strings are not common in app code but are mostly used in obfuscated libraries.

1.3. Structure of this Thesis

The remainder of this dissertation consists of four parts with chapters, which are organized as follows. Chapter 2 introduces the basic terminology for the remainder of the dissertation. First, it describes the different components, files, and code sections in an Android app. Second, it gives an overview of different obfuscation techniques, including the information which parts of an Android app are challenging to obfuscate. Finally, the chapter gives a brief introduction to basic machine learning techniques that we have used for our approaches.

Part I explains which approaches are necessary for the handling of obfuscated libraries. It begins with the explanation of the difference between library detection and separation of app code. Chapter 3 introduces our approach *LibDetect* to identify the libraries of in obfuscated apps. Afterward, the separation approach *AppSeparator* is introduced in Chapter 4. In Chapter 5, we introduce our repackaging detection approach *CodeMatch* and show the effectiveness of *LibDetect* and *AppSeparator* is shown by using them for repackaging detection.

The following Part II describes our library mapping approach *LibMapper* and our approach for the detection of obfuscated names *ObfusSpot*. Chapter 6 describes *LibMapper* that identifies obfuscated library classes using *LibDetect* and *AppSeparator*. Using both approaches, *LibMapper* determines for each identified class appropriate names for the class, its fields and its methods. Chapter 7 explains the different techniques *ObfusSpot* uses to identify an obfuscated name automatically.

Part III shows our string deobfuscation approach *StringHound*. While the Chapters 8 and 9 elaborate the manual and automatic identification of obfuscated strings, Chapter 10 explains the slicing and recovering of string values. During our study, we identified 21 unique string obfuscation techniques that use to design two detectors for obfuscated strings. *StringHound* uses these detectors and extracts the context of each obfuscated string to reveal the content of the string.

Finally, Part IV concludes this dissertation by summarizing the combined efforts of our deobfuscation approaches and discussing future directions.

For each approach, we present its evaluation in the chapter that describes its implementation.

1.4. Publications

Most of the contributions presented in this thesis have previously been published at high-quality venues of software engineering and security conferences. This section gives an overview of the author’s publications and how this thesis extends the respective parts of the publications.

The following lists security and software engineering publications that have been published by the author.

- [GMB⁺20] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonyamy, and Mira Mezini. "Hidden in plain sight: Obfuscated strings threatening your privacy". In Proceedings of the Asia Conference on Computer and Communications Security, ACM, 2020. accepted for publication.
- [GAE⁺17] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. "CodeMatch: Obfuscation Won’t Conceal Your Repackaged App". In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 638-648. ACM, 2017.
- [GSWH15] Leonid Glanz, Sebastian Schmidt, Sebastian Wollny, and Ben Hermann. "A Vulnerability’s Lifetime: Enhancing Version Information in CVE Databases". In Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business, page 28. ACM, 2015.
- [EHMG15] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. "Hidden Truths in Dead Software Paths". In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pages 474-484, 2015.

This thesis is based on the following publications and parts of the thesis may contain verbatim content from these publications.

CodeMatch: Obfuscation Won’t Conceal Your Repackaged App This publication [GAE⁺17] describes the different representations used in *LibDetect* and *CodeMatch*. We evaluated *LibDetect* and *CodeMatch* against state-of-the-art approaches, measured the ratio of repackaged apps in five different app stores and established an approach to filter out apps produced by app generators. Compared to the paper, this dissertation extends the discussion of work related to *LibDetect* in Chapter 3, and extends the evaluation of *CodeMatch* (Chapter 5) with the new app-code-separation approach from Chapter 4.

1. Introduction

Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy In this publication [GMB⁺20], we conducted a study of state-of-the-art string obfuscation techniques and developed an approach to identify and recover obfuscated strings. In comparison to the paper, in this thesis, we provide more details about the study’s string obfuscation techniques in Chapter 8, and extended the evaluation of the obfuscated string identification in Chapter 9 to compare our approach with other state-of-the-art techniques.

The content of the chapters 4, 6 and 7 appears for the first time in this thesis and will be submitted after this thesis.

Name Obfuscation Against Obfuscation Name Detection and Name Recovery Chapter 4 describes an approach to separate app code from library code to improve analyses that focus only on app code. In Chapter 6, we introduce *LibMapper*, an approach that outperforms the state-of-the-art name inference approach *DeGuard* [BRTV16] in recovering obfuscated names in libraries. Finally, we introduce, in Chapter 7, an approach that detects obfuscated names even if they were manipulated using dictionaries.

2. Background

In this chapter, we present the background information needed to understand the content of this dissertation. First, a general overview of the architecture of an Android app is given, followed by obfuscation techniques for Android apps. Finally, we provide an overview of several machine learning algorithms used in this dissertation's approaches and experiments.

2.1. Android Application Architecture

The Android operating system is used on smartphones, smartwatches, tablets, TVs, and various other devices. The apps installed by users range from games to managing finances or even remote controlling drones [and20c].

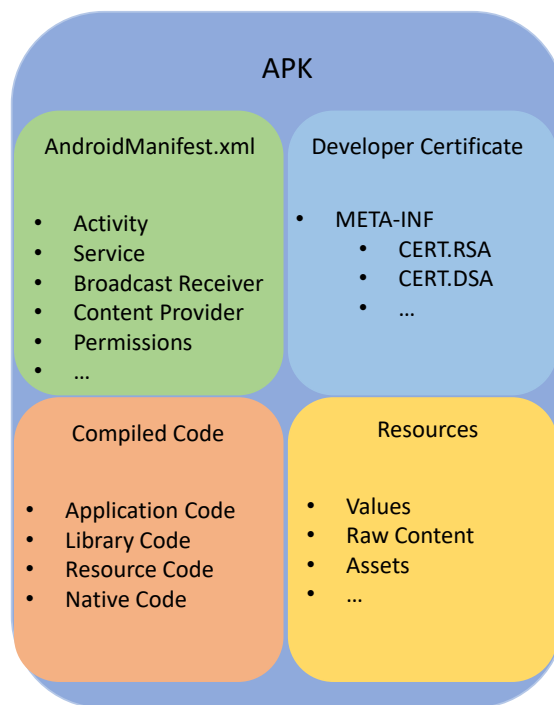


Figure 2.1.: Content of an APK

An Android application is shipped as an Android Package (APK) that contains different files [apk20]. It contains the Android manifest that specifies the application's

2. Background

starting point, the developer key used to sign the APK, the executable code, and the resources needed to run the application. All code includes the main application code, the code of the used libraries, and the native code. While the former two are compiled together into one file, the native code can be used from different other files. Figure 2.1 shows the content of an APK, which is used throughout this dissertation. The following describes the content in more detail.

2.1.1. Android Manifest

The Android build system stores its manifest in the `AndroidManifest.xml` file that also specifies the package name that Google uses as a unique identifier for the whole app. This package name also defines the namespace of the application [and20a]. Additionally, the manifest defines hardware capabilities, intent filters, icons, labels, resource values, required permissions, and code components. The code components are used as entry points of the app are divided into activities, services, broadcast receivers, and content providers. The following briefly describes the purpose of all the mentioned items in the Android manifest.

Activity: An activity provides the user with an interactive window that contains either a full-screen window or a floating window [Act20]. For instance, in activities, a user can access the camera to take a picture, press a button, write text into a text field, swipe the window, or perform other activities.

Content Provider: Content providers are essential building blocks of an application to share the encapsulated data among different applications [con20]. For instance, content providers present a method of selecting images and triggering the transfer of an image to another contact.

Service: A service executes long-running actions without interacting with the user or provides other applications with additional functionality [ser20]. For instance, after a user selects to share a picture using a content provider. A service transfers each byte of the picture to the contact's device in the background.

Broadcast Receiver: A broadcast receiver is a basic class that handles incoming broadcast requests. Such requests consist of intents that specify which action should be performed by which process. For instance, an app can use broadcast requests to access specific functionalities of other apps or exchange messages with them [bro20].

Permissions: The Android operating system uses permissions to protect the privacy of the device user. Developers can specify which permissions are needed to use their apps. Additionally, the developer may define permissions that grant other apps the usage of the provided services. Nevertheless, users need to decide whether they grant the permissions for an app they want to install. Since most users are not aware of the consequences for granting specific permissions, they grant almost every permission request.

The Android system differentiates between normal, signature, and dangerous permission categories to assist users in their decision process. These categories manifest in different authorization procedures: The system directly grants normal permissions without informing the user because they have little impact on the user's privacy. For instance, the user is not notified if an app accesses the internet because this permission was already granted at installation time. Signature permissions are automatically granted by the system if an app was signed with the same developer key as another app that already has these permissions. This category uses the fact that the required permission was already granted at the installation time of the app with the same signature and could be accessed via the other app. Dangerous permissions are only granted if users explicitly give their consent because these permissions interfere directly with the privacy of the user [per20]. For instance, the system asks the user if an app tries to send a SMS.

Icons, Labels & Resource values: For each app, the developer can set an icon and a label used as default if no other icon or label is specified. Resource values reference the location of additional icons, labels, and other values that cannot be hardcoded into the manifest file. They depend on external behaviors, such as the device screen, the language, or other [res20].

Intent Filters: Activities, services, and broadcast receivers communicate with each other by sending intents. Intents contain actions and the necessary data for the actions that the intent receiver should perform. If one of the above components activates an intent, the system directs the intent to the app that specifies, in its manifest, the corresponding intent filter [int20a].

Hardware Capabilities: A developer defines hardware capabilities to ensure that the device on which the user installs the app possesses these capabilities. For instance, an app that scans barcodes of products needs the camera to perform its function. As a result, the developer may specify the camera capability in the Android manifest by using the `uses-feature`-tag. In contrast to permissions, hardware capabilities do not define which resources should be accessed, but which hardware should be available to allow the app to execute the full functionality or only parts of it [and20a].

Developer Certificate Before developers can publish their apps to a public store or run them on any device, they need to sign them. For the signing process, the developers need to create a certificate [cer20], which they can generate using either the RSA [RSA78] or DSA [dsa20] algorithm. After generating and signing, the Android system can attribute a specific app to specific certificate owners. They can use signature permissions to access specific functionality without notifying the user again because the permissions have already been granted to another app signed with the same certificate.

2. Background

2.1.2. Executable Code

The executable code consists of the main application code, all used libraries, and the native code. While the two former code parts interact with the Android system via the Android class library (*android.jar*), the native code has direct access to the system [and20b]. Developers write the main application code in Java or Kotlin that the compiler bundles with the referenced dependencies and the used libraries into a Java Archive (JAR). The Android build system transforms the JAR file from the stack-based format of the Java Virtual Machine (JVM) to the register-based Dalvik Executable (DEX) format. The transformed file is often stored using the name `classes.dex`. It contains all application code, library code, and code to reference resources, but not the native part of the app. In the following, we explain all of the different code parts:

Application Code: The application code contains all components such as activities, services, broadcast receivers, and content providers under the namespace, which the developer specified in the manifest. However, in addition to the code under the namespace, the developer can also write code under all other namespaces, making it challenging to distinguish between the main code and library code.

Library Code: Library code enables the developer to use functionality without writing it from scratch. To use it, the developer needs to reference the code in the dependencies. An often-used library is the Android support library, which enables the developer to write code without taking care of the version of the underlining Android class library.

Resource Code: The developer can include a variety of resources into an app to improve the look-and-feel for users. Resources can contain animations, assets, pictures, colors, the structure of individual windows, fonts, values, and raw contents [res20].

For each resource category, the Android system generates a corresponding `R` class that contains the resource IDs for each referenced resource. The build system stores these resource IDs in `public static final` fields of these classes and in specified mapping files that can be used to load the contents of the resources.

Native Code: With native code, an app can perform tasks more efficiently or use additional native libraries [nat20]. For communicating with native code, the developer needs to use the Java Native Interface [jni20]. For instance, developers can use native code to improve the efficiency of their realtime games. The code is shipped in the APK but separated from the `classes.dex` file.

While APKs contain Dalvik bytecode, the tools described in this dissertation operate on Java bytecode. For this reason, we use *Enjarify* to transform the Dalvik code back into a JAR file. In contrast to other DEX-to-JAR transformers, *Enjarify* correctly handles Unicode names, constants, casts, and exception handlers [enj17].

2.2. Obfuscation Techniques

Code obfuscation manipulates the syntax of a given program without changing the semantics of the code [MKK07]. In contrast, data obfuscation tries to hide the content by changing or hiding its appearance. Since an app can only work properly with the original data, the data needs to be reversed before the app is executed. For this purpose, obfuscators integrate deobfuscation logic into the app, which reveals the original data before it is used [WHA⁺18]. Developers use both of the obfuscation techniques in benign apps to protect their intellectual property. However, malware writers also use these techniques to hide their malicious payload.

2.2.1. Obfuscation of Android Apps

This section presents common obfuscation techniques that are the subject of this dissertation’s analyses. Previous works [CTL98, UPSB⁺11, SLGL09] found these techniques in the wild, and some of them are used by known obfuscators [All19, Das19, Dex19c, Pro17, dex19a, ZWZJ14, str19]. We also include optimization techniques in our list because, akin to obfuscation, they transform the code so that it is harder to identify. Throughout this dissertation, we will refer to both techniques as obfuscation techniques.

Name Mangling: Meaningful identifiers, such as field, method, class, and package names, are replaced by meaningless, small sequences of characters; e.g., “**Person**” → “**aa**”. Furthermore, Allatori [All19] replaces meaningful names by case-sensitive sequences of characters. With this replacement, tools that operate on a case-insensitive filesystem (e.g., Windows) would overwrite all files with the same case-insensitive name. Therefore, these tools cannot analyze the overwritten files. Most obfuscators cannot only replace meaningful names by sequences of characters but also by words from a dictionary which replace meaningful names by sequences of these words [Pro17].

An obfuscator can replace package identifiers by other characters or even by an empty string. This change moves all classes of the package into the default package. While shortening the names makes code analyses harder, it also improves the app’s overall performance due to the smaller code size [Pro17].

Nevertheless, obfuscators cannot change all code-entity names because some methods inherit or overwrite methods from the Android class library. The obfuscation of these code entities would break the inheritance relation. Additionally, if obfuscators manipulate resources or code components referenced by external files, they must update these files. For instance, if an obfuscator manipulates the components specified in the Android manifest file or referenced resources in the code by resource IDs, it must update the manifest file, the resource files, and the code so that the app continues to work properly [WLG⁺17].

Modifier Changes: Field, method, and class modifiers can mostly be changed without affecting the program’s semantics. The modifications can range from raising

2. Background

the visibility of classes or class members to more complex ones. For instance, raising the visibility from `package private` to `public` or adding/removing the `final` modifier is straightforward, but transforming an instance method into a static one requires an extra parameter to make the `this` reference explicit. Deobfuscators [Jav20b] filter out methods with a `synthetic` modifier because they assume that a compiler generates these methods. For this reason, some obfuscators add this modifier to all methods in order to hide them.

Structural Changes to a Method’s Implementation: A fundamental technique is to add non-operational instructions such as NOP’s - i.e., instructions that do not affect the method’s semantics but modify the code’s syntactic structure. The primary effects are a larger method body and shifted targets of jump instructions, such as, `if`, `switch` or `goto`. These changes affect approaches that try to identity methods using checksums or cryptographic hashes as fingerprints of the methods. More progressive approaches change not only the syntax but also the structure of the methods. For instance, some approaches change the `if`-instruction (e.g., from “`if >`” to “`if ≤`”). This change affects the method’s structure because the positions of the basic blocks that are referenced by the `if`-instruction need to be switched.

Code Slicing: Most apps do not use all functions of the included libraries. Therefore, some obfuscators extract all the necessary code by gathering the essential functions using a technique called slicing [Wei81]. For instance, if the entire code does not call a method, it can be removed by creating a slice of the essential code. The removal of multiple methods can lead to the removal of entire classes if no code accesses these classes. Moreover, the removal of classes can lead to the removal of entire packages if the packages do not contain any class that is accessed by the main code.

Code Restructuring: Common obfuscators can move classes between packages and methods between classes and update the references to these code entities. Such changes affect all call sites related to the changed class structure. As in *Code Slicing*, the re-location of methods or classes can lead to the removal of entire classes or packages.

Method Parameter Manipulation: Reordering, removing, or adding method parameters affects both the method signature and its body. These changes generally require corresponding updates of all call sites. An obfuscator can reorder a method’s parameters if it rearranges the arguments that are passed to this method in the same order. The removing or adding of parameters requires the re-assignment of indexes to the load and store instructions of the calling methods and the manipulated method.

Constant Computation: An obfuscator can replace constant values with expressions that compute the constant. For instance, an obfuscator can replace the constant 100 by the computation `10*10`. The computation of constants can also include

the insertion of fake branches into the code. Such branches can either contain a condition that always resolves to the same value so that only one of the branches is used or consist of two branches that perform exactly the same computation. Simple analyzers cannot identify any of these fake branches [CTL98].

String Obfuscation: A more advanced technique is string obfuscation [Dex19c]. It can use encryption, encodings, and different representations to hide the content of a string. Obfuscators add logic at the code location that processed the original string to reveal the obfuscated string's content before it is processed [WHA⁺18]. For instance, the string "Blue" is replaced by the string "Oyhr", and during runtime, the app executes the integrated Rot13 [Sch07] deobfuscation logic directly before the app code needs to process the revealed string.

Fake Types: Already existing classes, in particular from libraries such as the Android class library, are duplicated or inherited and used within the program instead of the original class. For instance, instead of using the class `java.util.HashSet` an obfuscator can use a fake type `com.MySet`. In more advanced cases a field's primitive type is changed; e.g., from `int` to `long`.

While obfuscators can duplicate as much information as possible, they cannot completely remove all `extends` or `implements` relations of all classes inheriting from the Android class library. Thus, deobfuscators can use these relations to determine the original type.

Code Optimization: Code optimizations can also influence the code's structure whenever an obfuscator inlines methods like *Code Restructuring*, propagates values as *Constant Computation*, removes unused variables as *Code Slicing*, or modifies the control-flow as *Structural Changes to a Method's Implementation*. Code optimizations include typical peephole optimizations such as removing non-used instructions, replacing multiple instructions by semantically equivalent ones, using algebraic laws to simplify or reorder instructions, using special case instructions, and resolving the computation of addresses where possible.

Resource Manipulation: As described above, Android apps use various external resources. Some analyses use the hash sums of resource files to identify a particular app. However, obfuscators can change resources. For instance, an obfuscator can add pixels to a picture, change words in an external text, add sounds to audio files and frames to video files.

Apps use resources by loading them using references in their code that are generated by the Android system. For each resource that can be accessed by the code, the system inserts an `int` field in a specially designated resource class in the code (e.g., `R.style`). The system assigns the field a resource ID, and the field's name matches exactly the name of the accessible resource file. For instance, a field that enables access to the resource `main` would appear as follows:

```
public final static int main = 163458935; where 163458935 is the resource ID.
```


2. Background

Since these fields are a part of the code, obfuscators may use *Constant Computation* or *Name Mangling* to manipulate the values or names of the fields so that an analysis cannot read the names or follow the values of the resources. However, if an obfuscator changes the names of such fields, it also needs to change the resources' file names. The change of these file names, in turn, can lead to other errors, as the respective resources might be used in other files. Consequently, most obfuscators either do not change the names of the fields in resource classes or change them only if the developer of the app keeps track of the changes in other files.

Hide Functionality: Sophisticated obfuscators hide functionality by encrypting, recompiling, compressing, and virtualizing selected classes and adding code that reverses the effects at runtime [SLGL09]. These techniques are currently the most effective obfuscation techniques, but generally, slow down the app's execution time, need advanced knowledge of the app's internal structure, or require manual code changes. Hence, they are rarely used in practice.

2.3. Machine Learning

In this dissertation, machine learning algorithms are used for two different purposes. First, objects are grouped by their measurable properties, which are called features. These groups are called clusters, and the process to create these clusters is called clustering. Secondly, we use machine learning to assign classes to objects. For this purpose, an algorithm also uses features from objects. However, during classification, an algorithm learns the relationship between the features and one or more target classes, either given or determined by the algorithm. After the algorithms have learned either clusters or the relationship between features and classes, the learned states of the algorithms could be stored as so-called models. Afterward, the algorithms could load these models to process features from an unknown object as input and output to which cluster the object belongs or to which learned class the object can be assigned.

The following briefly introduces the machine learning algorithms [MD01] used in this thesis for clustering and classification.

Suppose we want to analyze different weather conditions to either cluster them or extract a decision from the data when children can or cannot play outside. We collect in Table 2.1 the features *Outlook*, *Humidity*, *Windy*, and past decisions to *Play* outside or not as a target class.

2.3.1. Clustering

For the clustering example, we only use the features *Humidity*, and *Windy* from Table 2.1 and try to identify clusters in the weather conditions. For some algorithms [kme20], the number of clusters needs to be specified before starting the learning process. However, since we do not know how many clusters are present in our data set, we use the expectation maximization [DLR77] (EM) clustering algorithm that can determine the number of clusters using *cross validation*.

Outlook	Humidity	Windy	Play
sunny	71	false	true
sunny	5	true	true
sunny	0	true	true
sunny	15	false	true
sunny	20	true	true
sunny	5	false	true
sunny	80	true	false
sunny	90	false	false
sunny	69	true	false
sunny	95	true	false
sunny	69	false	false
sunny	68	true	false
overcast	5	true	true
overcast	15	false	true
overcast	5	true	true
overcast	20	false	true
overcast	69	true	false
rainy	20	false	true
rainy	50	false	true
rainy	30	false	true
rainy	80	true	false
rainy	90	true	false

Table 2.1.: Data set for playing outside

Cross validation is a statistical measurement method to determine the performance of an algorithm. Like other algorithms, the EM algorithm uses 10-fold cross-validation, in which the data set to be processed by the algorithm is randomly divided into ten subsets, and the algorithm is applied ten times to the entire data set. Typically, for classification, nine of the ten subsets are used to train a classifier, and one subset is used for validation. However, the EM algorithm uses all ten subsets to initially assign all data points to a randomly selected cluster and determine whether all data fits into one cluster. Afterward, if the data does not fit into one cluster, the algorithm increases the cluster number and repeats the clustering process until the data fits in the clusters. In our example, the algorithm uses *Humidity* and *Windy* features from our data to cluster them in three different clusters that correspond to the *Outlook*-Feature in 59.10% of the cases. Finally, the EM algorithm returns per cluster for each feature the mean value and standard deviation, which can be used to match new data points into the clusters or measure the distance of data points to the centers of each cluster.

2. Background

2.3.2. Classification

For the classification of data sets, our experiments analyze different classifiers to identify which classifier offers the best performance in terms of correctly classified objects. For these experiments, we investigated the classifiers *Multilayer Perceptron*, *Naive Bayes*, *Logistic Regression*, *Random Tree*, *Random Forest*, and *REP Tree* from Weka [HFH⁺09]. Weka is a framework used in many different research papers [Kun14, MJ15, AL11], which presents a variety of different clustering and classification algorithms. We chose the five classifiers because they are intended to represent all categories of Weka’s classification capabilities.

In the following, we explain the different classifiers using from Table 2.1 the columns *Outlook*, *Humidity*, and *Windy* as features and *Play* as a target decision to be learned. First, we introduce the three decision trees *Random Tree*, *Random Forest*, and *REP Tree*, followed by the algorithms *Naive Bayes*, *Multilayered Perceptron*, and *Logistic Regression*.

A decision tree consists of leaves representing the consequences of various decisions and decision nodes that contain conditions that determine the progress of a decision [Qui96]. Given the data in Table 2.1, a possible decision tree might be constructed like the one in Figure 2.2. The decision tree can use each feature in multiple decision nodes and increase the likelihood of deducing a definite prediction. Continuous values like *Humidity*, in our example, are divided by the decision trees into different value ranges to find the most suitable condition for a decision node.

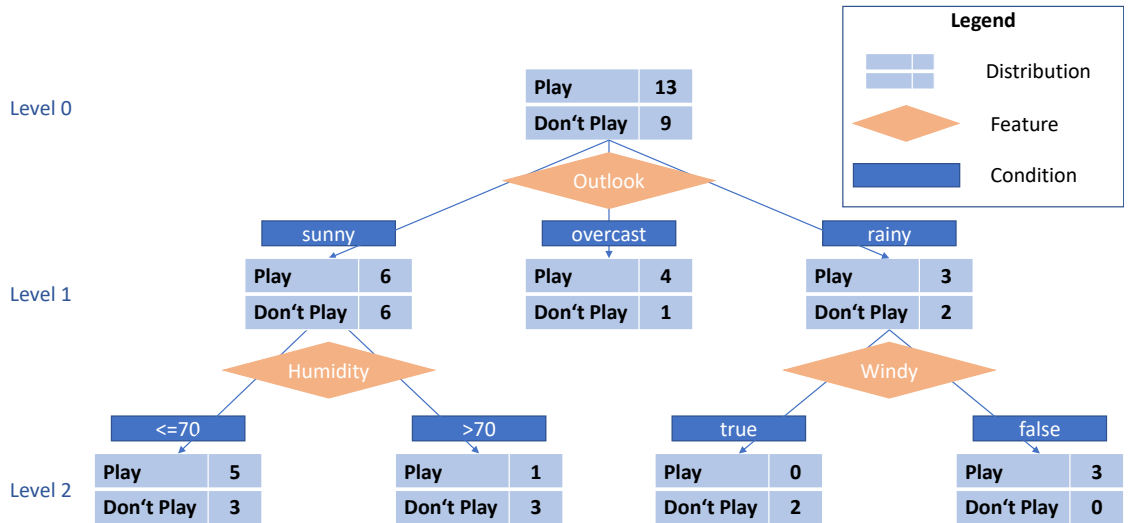


Figure 2.2.: A possible outcome of a decision tree [dec20]

REP Tree The REP Tree algorithm [rep20] constructs a decision tree by identifying features that represent the best splitting point between the target classes. Those features are positioned in nodes that are further up in the decision structure. After a feature is

selected, the algorithm searches for the best condition to split the data. For instance, the algorithm decided that *Outlook* is the top feature in Figure 2.2. Based on this decision, the algorithm could combine the possible values of *Outcome* in different variants such as {sunny, overcast}, {overcast, rainy}, or {sunny, rainy}. However, it decided that all values should be split into separated decision nodes because it leads to an early decision for the value overcast.

After the algorithm determined the feature and the first splitting point, it repeats this process until the remaining data is allocated into their target classes. If all remaining data has the same target value, it adds a leaf representing it. In contrast, if all remaining features values cannot be further split, the algorithm uses each target value's probability. For instance, in Level 2 of Figure 2.2, each node of the *Windy*-feature contains only data that belongs to one target value. In contrast, in Level 1, the node of *overcast* returns the probability of each target value.

A REP Tree can be constructed fast, and in contrast to other classifiers, using Weka's source code generation, it can be directly integrated into Java or Scala code.

Random Tree The Random Tree algorithm [Bre01] builds a decision tree from a given data set using a similar approach as the REP Tree algorithm. However, instead of determining the best feature for a decision node, the algorithm randomly selects one of k features at each node. This procedure allows an even faster construction of a decision tree by reducing large feature dimensions to the number of k features. For instance, this algorithm could use only the feature *Outlook* and would calculate for each decision node the probabilities of the possible target values similar to Level 1 in Figure 2.2. Thus the algorithm's decision for the value *sunny* would entirely depend on the chosen threshold value.

Random Forest While the Random Tree randomly selects from a set of features in each decision node, a Random Forest constructs multiple such trees and uses the ensemble of these trees to predict the target value [Bre01]. It calculates the target values for all trees and uses the value that is most frequently suggested.

For instance, the Random Forest could build different Random Trees from each possible combination of features in Table 2.1 and compose the trees for the classification. Let us assume that the Random Forest builds a single tree for each feature, calculates the possible probability values, and composes them. Such a Random Forest would produce the following outputs for the last row of Table 2.1: The *Outlook*-tree would output for *rainy* *Don't Play*, the *Humidity*-tree would return for the value > 70 *Don't Play*, and the *Windy*-tree would return also *Don't Play*. The most frequent result corresponds to the target value *Don't Play*, which is returned by the Random Forest.

Naive Bayes Contrary to the above-described decision trees, the Naive Bayes algorithm [JL95] determines a target decision by calculating the probabilities (Pr) of each target value (A) and all feature values (B). Afterward, the Bayes theorem [Bay91] from Equation 2.1 is used to predict based on the probabilities a result for an unseen data

2. Background

	Play Outside				
	true	false	P(true)	P(false)	total
sunny	6	6	6/13	6/9	12/22
overcast	4	1	4/13	1/9	5/22
rainy	3	2	3/13	2/9	5/22
total	13	9	13/22	9/22	100%

Table 2.2.: Probabilities for playing outside using the Outlook-feature

point. For instance, the algorithm would determine for each feature in Table 2.1 the number of times the target value is *Play* or *Don't Play* and divide this number by the overall number of *Play* or *Don't Play* rows. If we calculated the probabilities only of the *Outlook* feature, we would get the results in Table 2.2.

$$\Pr(A|B) = \frac{\Pr(B|A) \Pr(A)}{\Pr(B)} \quad (2.1)$$

Applying the Bayes theorem on the *Outlook*-feature produces the following results:

$$\begin{aligned} \text{Play}|\text{sunny} &= \frac{\frac{6}{13} * \frac{13}{22}}{\frac{12}{22}} = \frac{6}{12} = 0.5 \\ \text{Don't Play}|\text{sunny} &= \frac{\frac{6}{9} * \frac{9}{22}}{\frac{12}{22}} = \frac{6}{12} = 0.5 \\ \text{Play}|\text{overcast} &= \frac{\frac{4}{13} * \frac{13}{22}}{\frac{5}{22}} = \frac{4}{5} = 0.8 \\ \text{Don't Play}|\text{overcast} &= \frac{\frac{1}{9} * \frac{9}{22}}{\frac{5}{22}} = \frac{1}{5} = 0.2 \\ \text{Play}|\text{rainy} &= \frac{\frac{3}{13} * \frac{13}{22}}{\frac{5}{22}} = \frac{3}{5} = 0.6 \\ \text{Don't Play}|\text{rainy} &= \frac{\frac{2}{9} * \frac{9}{22}}{\frac{5}{22}} = \frac{2}{5} = 0.4 \end{aligned}$$

As we can see from the calculations, the Bayes theorem's application produces the same output as the split on Level 1 of the decision tree in Figure 2.2.

Logistic Regression The Logistic Regression algorithm [LCVH92] calculates a regression function based on the training data. Afterward, it uses a Sigmoid function [MD01] with a predefined threshold value to represent the target decision in binary form. Since regression can only be performed on numeric features, the algorithm converts all possible nominal features into binary ones. For instance, the values of the *Outlook*-feature from Table 2.1 are converted to create three features (*Outlook=sunny*, *Outlook=overcast*, and

Outlook=rainy). After the learning procedure, the algorithm returns a function that can predict a target value.

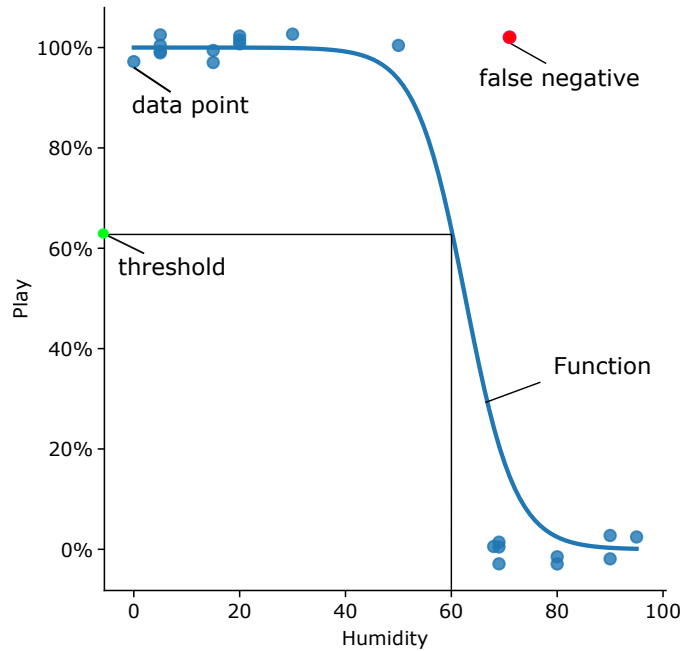


Figure 2.3.: Plot of the Sigmoid function for the Humidity feature

To illustrate a simple logistic regression, Figure 2.3 presents the function calculated for the *Humidity*-feature with the target decision *Play* from the Table 2.1. This illustration shows that the smaller the humidity, the more likely the children are going to play outside. Depending on the chosen threshold, the function produces more true positives or true negatives. In this example, it seems that a threshold above 60% of the y-axis would increase the function's predictive performance because every data point above the humidity of 60% is considered to result in the *Don't Play* decision. Consequently, only one false negative at the right upper corner remains.

Multilayer Perceptron The Multilayer Perceptron algorithm (MLP) [HTF09] belongs to the algorithms of neural networks. It arranges artificial neurons that learn processes similar to real neurons in a nervous system. It mainly uses so-called perceptrons, which correspond to neurons that only connect to forward directed neurons without back-propagation. These perceptrons are arranged in layers so that one layer of neurons communicates with the next layer to learn weights based on the input data.

The input layer is composed of as many perceptrons as input features. This layer is usually connected to one or more hidden layers that are transitively connected to the output layer. The output layer possesses as many perceptrons as output decisions are expected. All perceptrons are trained multiple times by processing the training data in multiple iterations to learn a function that converts input features to output decisions.

2. Background

The iterations are also known as epochs. The longer the algorithm iterates over a data set, the better it predicts the relationship between the features and the targeted decisions; however, longer training times make the learned model less suitable for predicting new data (overfitting). Given the data in Table 2.1, the MLP algorithm would first convert the nominal features into binary ones like the Logistic Regression algorithm and then create as many input perceptrons as the number of features (after conversion, we would get five features). Furthermore, the algorithm creates an output perceptron and as many hidden perceptrons as selected. For instance, for the data from Table 2.1, a Multilayer Perceptron with a hidden layer of 12 perceptrons would be connected similarly to the one in Figure 2.4.

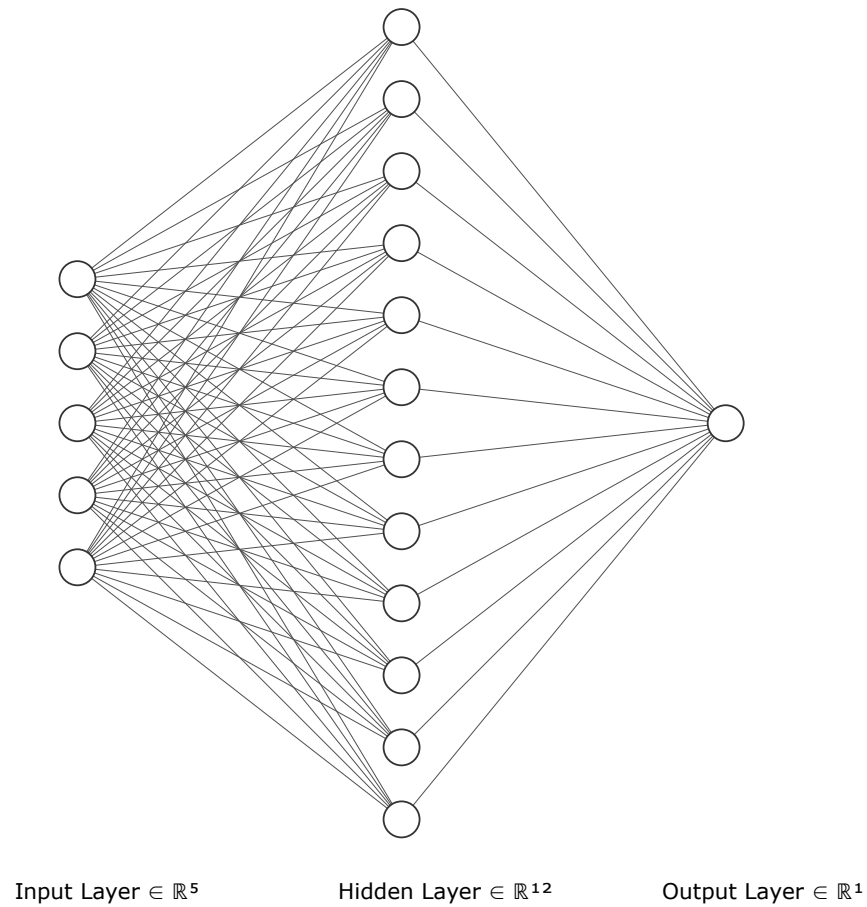


Figure 2.4.: Example Multilayer Perceptron for the weather data¹

¹Source: <http://alexlenail.me/NN-SVG/index.html>

Part I.

Obfuscated Libraries

Obfuscated Library Detection and Separation of App Code

The sharing of code components as libraries positively impacts the development speed of new software products [Boe99]. Functionalities condensed in these libraries no longer have to be written from scratch, which led to the fact that, on average, more than 60% of the sub-packages in each Android app belong to library code [WGMC15].

Nevertheless, many analyses need to filter out library code to run efficiently or focus solely on application code [YFM⁺17, IWAM17, STDA⁺17, and17, LLB⁺17]. For instance, analyses of intellectual property may focus solely on application code to identify plagiarism and other license violations. However, the separation of application code is challenging since the Android compiler bundles library code with the application code in a single binary. Moreover, many developers obfuscated their binaries to reduce the used disk space and execution time, so that the separation of application code becomes even more laborious [pro20a, Don18].

While the separation of app code is already challenging, some analyses require the detection of libraries. For instance, the detection of vulnerabilities and comparison of similar functionality needs identifying all used library versions [BBD16]. If an app contains obfuscated library code, an analyst cannot directly compare it with a library version that contains a known vulnerability. Without identifying obfuscated library versions, the vulnerabilities remain on the user's device and may corrupt the user's privacy and security.

Previous works studied library detection and the separation of app code using different techniques and for different platforms [CLZ14, ZZJN12, LWW⁺17, MWGC16, BBD16, ZDZ⁺18, WWZR18, FR19, ZBK19, CWC⁺16, STA⁺16, TLS17, HLT18]. For instance, current malware analyses [gui17], in low-level code, disassemble the machine code and identify library functions by mapping the first 32-byte of a function with a white list of previously collected bytes of library functions. Further analyses use white lists or heuristics to identify library packages [CLZ14, ZZJN12]. However, neither the white lists nor the heuristics are sufficient to identify most of the library code.

Due to the lack of effective library detection and app code separation, analyses that require these techniques cannot do their work properly. For instance, not identified library code may cause apps are considered as repackaged apps only because they use the same library code. In this part, we present two approaches: One for detecting libraries and one for separating app code. Both approaches are used by our repackaging detection to filter out libraries in apps of different sizes.

In Chapter 3, we introduce our approach *LibDetect* that identifies libraries contained in Android apps by using five representations. These representations assist *LibDetect* in

identifying libraries as precisely as possible by abstracting only over as much information as necessary.

Chapter 4 describes our *AppSeparator* that separates app code from library code. For the separation, we developed a machine learning model based on features derived from the assumption that library code does not access application code but vice versa. We base this assumption on the fact that libraries are available as stand-alone artifacts without depending on any application code [AL12], and, therefore, cannot access it.

To show the usage of library detection and app-code separation, we describe in Chapter 5 our approach *CodeMatch* that identifies repackaged apps by comparing the similarities of two app codebases. For the similarity measure, *CodeMatch* uses *LibDetect* and *AppSeparator* to filter out library code that would otherwise bias the similarity results of the approach.

In the end, we summarize all contributions of this part.

3. Library Detection

Developers of Android apps use libraries to extend the spectrum of services for their users. However, these libraries become outdated over time and may contain vulnerabilities. Some developers may not be aware of this process, as certain libraries are only available in obfuscated form. If analysts try to examine such an app, they need to detect each library before proceeding with any further analyses. Consequently, library detection is one of the first steps taken by analysts to get an overview of an application’s capabilities and features. For instance, library update, vulnerability, plagiarism, license violation, and repackaging detection can use library detection as a preliminary analysis to identify or filter out libraries.

Since the manual detection of large amounts of library code is not feasible, analysts often use white lists of library package names to identify their apps’ code. While this procedure may save analysis time and reveal some parts of a library, it is not suitable to identify library packages with obfuscated names. For instance, as described in the background, *Name Mangling* transforms library package names so that they are no longer distinguishable from app packages.

As a result, more advanced library detection approaches use checksums [gui17] of application and library classes to distinguish between them. However, a single bit-change in such a checksum can evade such approaches. Since most apps contain obfuscated libraries [Don18] to protect their intellectual property or reduce the required disk space, bit-changes occur frequently and render simple checksum comparisons ineffective.

Further approaches [LWW⁺17, MWGC16, BBD16, ZDZ⁺18, WWZR18, FR19, ZBK19] build fuzzy signatures of methods or classes that remove or replace information of a method or class. However, these approaches combine the extracted fuzzy signatures to entire packages or use package information in the signature, which makes them vulnerable to *Code Restructuring*, *Code Slicing*, and *Code Optimizations*.

To address the drawbacks of previous approaches, we propose a new library detection approach called *LibDetect* that uses representations of method bodies. Because classes contain obfuscated code, we establish a reliable comparison of method bodies using several hierarchically organized representations. The first one consists of the original bytecode. Each following representation abstracts over additional aspects, such as the used identifiers or the control flow to match method bodies with higher obfuscation complexity. Hence, each higher level is less precise but potentially enables a higher recall.

As a result of the precision/recall trade-off, we internally organized our library detection to use the representations step-by-step. If an obfuscator only marginally changed a library method, *LibDetect* will identify the method using less abstract representation than more effectively obfuscated methods. After identifying the library method bodies,

3. Library Detection

LibDetect regroups them to match the potential original library classes. This procedure enables the identification of methods and classes even if an obfuscator moved these across class/package boundaries.

In order to evaluate the robustness of our representations, we extract roughly 200 APKs from Maven Central [Mav17], which represent samples of libraries and obfuscate them with a state-of-the-art obfuscator (*DexGuard* [Dex19c]). Afterward, we tried to reidentify the original library methods. For the evaluation of *LibDetect*'s competitiveness, we randomly selected 1,000 apps, identified the libraries manually to establish a gold standard, and compared the precision and recall of *LibDetect* against LibRadar [MWGC16] and a white-list approach.

The remainder of this chapter describes state-of-the-art library detection approaches in Section 3.1, our approach *LibDetect* in Section 3.2, the evaluation in Section 3.3, threats to the validity of our experiment in Section 3.4, and conclusions in Section 3.5.

3.1. State-of-the-Art Library Detectors

In this section, we describe related work for our library detection approach. First, we describe approaches that use simple white lists to filter out library code, followed by more advanced techniques.

White Lists Different library detection approaches [CLZ14, ZZJN12] use common library *white lists* to detect and filter out library code. White lists contain package names of known libraries, and several approaches compare them with package names contained in Android apps. Currently, the most extensive white list is collected by Li Li et al. [LKLT⁺16]; it contains over 5,000 different names of library packages. The problem with using white lists is that if an obfuscator changes one character of a library's package name, it can completely evade the library detection.

LibD [LWW⁺17] uses the sub-/super-package relation (*inclusion*) and the inheritance relation between classes across packages (*inheritance*) to construct one reference graph per library. The tool compares these graphs with graphs extracted from an app. *LibD* constructs a graph using (sub-)package names as nodes and inheritance or inclusion relations as directed edges.

While this approach reduces the information needed for comparing libraries to the package level, it is vulnerable to changes that split or merge packages (i.e., *Code Restructuring*). If the package hierarchy is changed, the graph has a different number of edges per node, and, therefore, *LibD* is no longer suitable for the detection of libraries.

LibRadar [MWGC16] detects libraries by extracting for each package a feature vector consisting of the observed Android API calls and using LSH-hash [RU11] as a fingerprint for the comparison of these package vectors.

3.1. State-of-the-Art Library Detectors

This approach is resilient against the renaming of packages, but cannot handle *Code Restructuring*, *Code Slicing*, and *Code Optimization* because all these obfuscation techniques merge or split packages, which can affect each package vector.

LibScout [BBD16] identifies obfuscated library versions by using the Android API. It generates Merkle-tree-hash [Mer87] profiles in three steps to identify library code. First, it extracts all method signatures from an app, removes the method names, and replaces all types of a method signature that do not belong to the Android API with an “X”. E.g., the method signature `MyHash addHashObject(int,String,HashBase)` is transformed to `X (int,String,X)`. Afterward, it hashes the new method representations using a cryptographic hash.

Second, *LibScout* sorts all representations of a class and hashes them together (class hashes).

Third, it sorts the class hashes again and hashes them per package (package hashes). Finally, it uses all hashes to identify library code depending on the obfuscation of methods, classes, or packages. If the package hash does not match, *LibScout* uses the class and the method hashes to identify a library fragment. If no hash matches, the code element does not belong to the library code. The hashes are generated for different library versions to identify specific versions.

Since the computation of the tree hash inherently reflects the implicit tree structure between packages, classes, and methods, *LibScout* is not robust against cross-class/-package *Code Restructurings*. For instance, it would fail to identify all classes that belong to a library because of the assumption that an obfuscator moved all classes into the same package. Additionally, it is vulnerable against *Method Parameter Manipulation* since its method representation does not handle reordering, addition, or removal of parameters.

LibPecker [ZDZ⁺18] identifies library classes by extracting different information from the classes, their fields, and their methods. From all code entities, it extracts a selected set of access flags, the package membership, all used types from the Android class library, and the array dimensionality. Afterward, *LibPecker* compares all information extracted from a library with information from an app by considering the number of members. If the information of a library class matches a class the app, it is flagged as a library class.

While *LibPecker* can use the extracted information to identify light obfuscated classes, it is still sensitive to *Modifier Changes* since the selected access flags such as `static` are still forgeable by obfuscators. Additionally, it is vulnerable to *Code Slicing* and *Code Restructuring* because the replacing or removal of methods or classes can cause massive changes in the extracted information. Consequently, *LibPecker* misses library classes that are scattered across different packages or moved into one package.

Orlis [WWZR18] identifies library classes by using Merkle-tree-hashes as *LibScout*. However, *Orlis* does not only use the method signature, but it also extracts all transitive method calls of a method, and represent them in the same way as the method representations from *LibScout*. Afterward, it sorts the new representations and hashes all

3. Library Detection

representation of a class or app using a rolling hash algorithm to compare the resulting hashes with ones from apps or their classes. *Orlis* uses the algorithm to immunize the fuzzy representation against small code changes.

However, unlike our approach, *Orlis* is vulnerable to *Code Restructuring* and *Code Slicing* because the replacement or the slicing of large methods changes their fuzzy representations of classes. Additionally, *Orlis* is vulnerable to *Method Parameter Manipulation* because the usage of *LibScout*'s signatures is not safe against reordering, addition, or removal of parameters.

Lastly, a significant threat for the *Orlis* approach is *String Obfuscation* because string obfuscation adds calls to new deobfuscation methods to each class. Since all methods containing an obfuscated string call the deobfuscation methods, *String Obfuscation* changes large proportions of *Orlis*'s method representations.

Feichtner et al. [FR19] propose an approach that extracts features from the abstract syntax tree (AST) of all library methods and compares these with the ones in apps. The tool builds a vector for each AST by counting the occurrences of parameters, local variables, and method invocations. Afterward, it uses vectors from library classes and packages to match app classes and packages. The matching is performed by calculating the inclusion of the vectors in library classes and packages.

Unlike our approach, the tool is vulnerable to *Code Slicing*, *Code Restructuring*, and *String obfuscation* because they change the counted instructions in the AST vectors. Additionally, the tool is sensitive to *Method Parameter Manipulation* because the removal or addition of parameters also changes the AST vector. For instance, we could remove all parameters and replace them with field accesses. This action would completely change the vector.

LibID [ZBK19] builds method representations by combining class access-flags, class inheritance-information, method descriptors, and an abstract form of basic blocks. It extracts these method representations from all libraries and compares them with all extracted representations of apps. Afterward, *LibID* compares the matched method signatures to known library class and package constellations. For method parameter-types that can be changed by obfuscators, it replaces each type that is not from the Android class library with a **X** as done by *LibScout*.

In contrast to our approach, *LibID* is vulnerable to *Modifier Changes* since their method signature contains the class access flags. It is also vulnerable to *Code Slicing* and *Code Optimization* because both techniques can remove certain information and change the representation of methods. Additionally, *Code Restructuring* can move methods and classes to different classes and packages, which may break *LibID*'s class and package constellation matching. Furthermore, *Method Parameter Manipulation* can reorder, remove, or add parameters to the method descriptor, which evades *sLibID*'s representation-matching algorithm.

LibFinder [CWC⁺16] extracts from each app method, a control-flow graph (CFG), and calculates a centroid for each CFG. Physicists use centroids to describe the perfect point to balance any given shape. To identify libraries, *LibFinder* compares all centroids of a known library package with all packages in an app. The comparison is only performed if a library and app package share a common name prefix.

Because of the usage of the name prefix, *LibFinder* is vulnerable to *Code Restructurings*. If an obfuscator moves either some classes into another package or flattens the entire package hierarchy, *LibFinder* can no longer identify libraries. Additionally, since *LibFinder* calculates centroids for methods with bodies, it may miss interface and abstract classes of libraries.

LibSift [STA⁺16] identifies libraries in a given app by excluding the primary module of the app. For the exclusion, it extracts all packages, their hierarchy, and their dependencies. As dependencies of a package, *LibSift* uses method calls, class inheritance, and field references. Given the hierarchy and dependencies of all packages, *LibSift* merges the packages with high cohesion and low coupling to identify the app's primary module.

While *LibSift* needs no database of library packages, it is vulnerable to *Code Restructurings*. For instance, if an obfuscator merges all packages into one or moves many of the classes into another package, *LibSift* is no longer suitable to extract the primary package.

Ordol [TLS17] abstracts method instructions, calculates n-grams of the abstracted instructions, and matches these n-grams with the ones of known libraries. After the method matching, *Ordol* matches the surrounding classes with library classes. If the matched methods or classes violate some predefined conditions of *Ordol*, it refines the method matching until the methods and classes meet these conditions. In the end, *Ordol* outputs a coverage measure and its similarity score for each library.

Due to the usage of solely the method body as a similarity measure without considering the method signature, *Ordol* may fail to identify all classes that belong to a library. For instance, methods of interface and abstract classes do not contain instructions, and therefore, *Ordol* misses these classes. Additionally, the instructions' abstraction does not consider the rearrangement of parameter-lists from method calls, which may lead to missing methods and a less accurate library detection. *String Obfuscation* can also be the cause of method mismatches since it inserts additional methods and method calls, which may decrease libraries' similarity score.

Hongmu Han et al. [HLT18] propose a similar approach as *LibScout* except that they base their initial method signatures on the control-flow graph of a method. Despite the different initial signatures, their approach is also vulnerable to *LibScout*'s drawbacks that make their approach no longer suitable if an obfuscator uses *Code Restructurings*.

Discussion Table 3.1 shows a summary of each approach, its representation, and its weaknesses against obfuscation. The above discussions and the summary indicate that

3. Library Detection

Table 3.1.: Categories of library detection approaches

Approach	Representation	Name Mangling	Code Restructuring	Code Slicing	Code Optimization	Modifier Changes	Method Parameter Manipulation	String Obfuscation
White Lists [CLZ14, ZZJN12]	Package Names	✗	✗	✗	✓	✓	✓	✓
<i>LibD</i> [LWW ⁺ 17]	Package Inclusion/Inheritance	✓	✗	✓	✓	✓	✓	✓
<i>LibRadar</i> [MWGC16]	API Calls	✓	✗	✗	✗	✓	✓	✓
<i>LibScout</i> [BBD16]	Merkle-tree-Hashes	✓	✗	✓	✓	✓	✗	✓
<i>LibPecker</i> [ZDZ ⁺ 18]	Code Entity Info.	✓	✗	✗	✓	✗	✓	✓
<i>Orlis</i> [WWZR18]	Transitive Method Calls	✓	✗	✗	✓	✓	✗	✗
Feichtner et al. [FR19]	AST Vectors	✓	✗	✗	✓	✓	✗	✗
<i>LibID</i> [ZBK19]	Basic Blocks	✓	✗	✗	✓	✗	✗	✓
<i>LibFinder</i> [CWC ⁺ 16]	Centroids	✓	✗	✓	✓	✓	✓	✓
<i>LibSift</i> [STA ⁺ 16]	Package Dependencies	✓	✗	✓	✓	✓	✓	✓
<i>Ordol</i> [TLS17]	Method N-Grams	✓	✓	✓	✓	✓	✗	✗
Han Hongmu et al. [HLT18]	Control-Flow Graph	✓	✗	✓	✓	✓	✓	✓

none of the existing approaches can handle all obfuscation techniques. Consequently, we need a better library detection that handles library instances with a changed package hierarchy, string obfuscation, or other advanced obfuscation techniques. Additionally, most of the approaches have either a missing database or are generally not publicly available. Beyond white lists, *LibRadar* is the only approach that is entirely publicly available at the moment. Both white lists and *LibRadar* have their specific limitations mentioned above. These limitations result in a poor recall, as we will show in our empirical comparison of these techniques against our new approach (cf. Section 3.3).

3.2. LibDetect

In this section, we present our approach *LibDetect* that enables the detection of library code from a given Android package (APK). *LibDetect* relies on abstract representations of the app’s code to handle different obfuscation techniques.

LibDetect is a code-signature-based library detection approach that can detect a library even if an obfuscator slices it down, moves its classes to another package, or puts instances of app classes into the packages of the library. Furthermore, *LibDetect* ad-

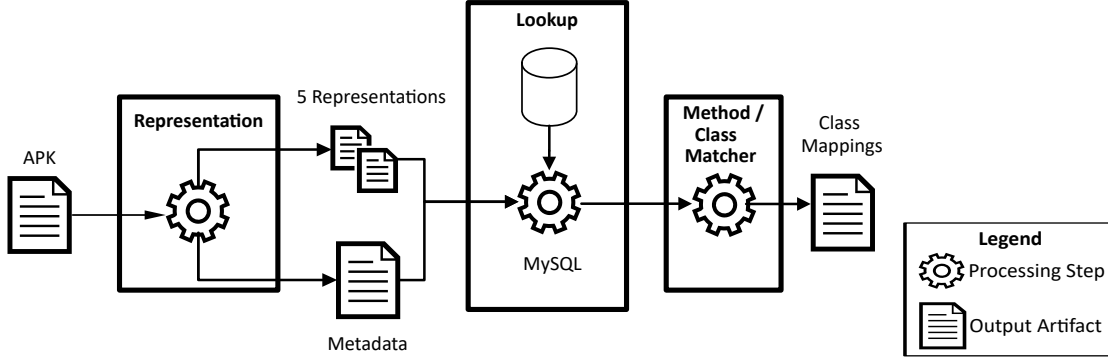


Figure 3.1.: Toolchain to detect library classes

addresses the shortcomings of existing approaches [CLZ14, ZZJN12, LKLT⁺16, MWGC16] that only identify libraries based on their packages. Such an approach may miss individual library classes or accidentally flag app classes as library code if it considers all classes in a package as library code. In contrast, *LibDetect* searches for copies of library methods and later aggregates potential matches to classes, i.e., *LibDetect* detects library code at the granularity of classes.

To match individual methods, we need to deal with the obfuscation of both methods and classes, because other methods may reference these. Since obfuscation introduces variation in the method’s code, we counter its effects by matching our abstractions from library code stored in our database. In this process, the degree of abstraction becomes a tradeoff between precision and recall: If *LibDetect* uses a more concrete representation to match a library method, it gets fewer or no potential matches than with more abstract representations. Additionally, matches with the more concrete representation yield library methods with higher similarity to the matched method in the app. However, with this representation, we might miss library methods changed by a more advanced obfuscator. Using a higher abstracted representation, we are more likely to find potential matches for an app method. Nevertheless, we may also falsely match the method with a larger number of library methods. Figure 3.1 depicts the overall process of *LibDetect* which we describe in the following sections.

3.2.1. Representation

In order to deal with the obfuscation of methods and classes (see Section 2.2), we use five different abstract representations of methods. The representations build upon one another, each abstracting over some additional elements of the original bytecode compared to its predecessors. Our approach uses the representations in a sequence, starting with the original bytecode to more abstract ones up to the most abstract. In this process, we consider only the findings of the least-abstract representation as our result. The following explains the generated representations in more detail:

We use the **Bytecode (BC)** of a method with its opcodes, the method signature, and offset, to reliably detect non-obfuscated library methods.

3. Library Detection

In the **Addressless Representation (AR)**, we remove all method modifiers, program counters, and NOPs, since an obfuscator can use this information to manipulate a method’s body. Additionally, we abstract over jump targets by replacing forward jumps with the key “*along*” and backward jumps with “*back*”. Taken together, this addresses respective *Structural Changes to a Method’s Implementation* and *Modifier Changes*.

In our example method, the column *AR & NR* in Table 3.2 shows that the references of the `if_icmpne →9`, `if_icmpge →18` and `goto →19` from column *BC* are changed to `along` and the program counters are dropped.

In the **Nameless Representation (NR)**, we address *Name Mangling* and *Fake Types*. To construct the representation, we remove the method names from method signatures and invocation instructions. Additionally, we replace non-Android-API type references in return, parameter, field, array, and invocation instructions by lists of the types’ Android-API supertypes. These lists represent those parts of type information that an obfuscator cannot manipulate. We obtain them by traversing up the type hierarchies, collecting all interface and class types defined in the Android/Java SDK, including the type `java.lang.Object`. Given the type lists, we order all types alphabetically to avoid mismatches based on different type orders.

Table 3.2.: A `compare(int, int)` method in BC, AR, NR, and SPR.

BC	AR & NR	SPR
0:iload_0	iload_0	load
1:iload_1	iload_1	load
2:if_icmpne→9	if_icmpne→along	if→along
5:iconst_0	iconst_0	const
6:goto→19	goto→along	if→along
9:iload_0	iload_0	load
10:iload_1	iload_1	load
11:if_icmpge→18	if_icmpge→along	if→along
14:iconst_m1	iconst_m1	const
15:goto→19	goto→along	if→along
18:iconst_1	iconst_1	const
19:ireturn	ireturn	return

Table 3.3.: A method signature’s AR and NR.

	AR	NR
Declaring Type	MyHashSet	[HashSet, Object, Set]
Method Name	get	
Parameter	Key	[Object]
Return Type	int	int

Table 3.3 shows the AR and NR method signature of `Object get(Key)` which is declared by the app class `MyHashMap`. This app class is a clone of Android’s `HashMap` class. For this example, we assume that the app class `Key` inherits only from `Object` and that `MyHashMap` inherits from `AbstractCollection`, `Map`, and `Object`. While the signature of AR contains the app-specific type-information, its NR is identical to the `get` method from Android’s `HashMap`. This allows the representation to detect `MyHashMap` as a *Fake type*.

In the **Structure-Preserving Representation (SPR)**, we sort the parameter type lists from NR in lexicographical order to address *Method Parameters Manipulation*. Additionally, we avoid *Fake Types* by removing all type information and the indexes from load and store instructions. For example, the instructions `astore` and `dstore_2` for storing an `object` or a `double`, are both represented by `store`. In order to unify size-dependent instructions, such as `ldc` and `ldc_w` and we also drop all string constants, e.g., log messages, to address *Constant Computations* that exchange these. Furthermore, we improve our handling of *Structural Changes to a Method’s Implementation* by representing all compare and jump instructions by `if`. However, the jump direction, i.e., “along” or “back” is kept.

To provide an exhaustive mapping from bytecode instructions to their SPR, we added the mapping in the appendix of this dissertation (cf. Section 12.8).

In the **Fuzzy SPR**, we address stronger *Code Optimizations*, *Constant Computations* and *Code Slicing* by using fuzzy hashes of the token sequence from our SPR with the tool *SSDEEP* [Kor06]. This tool enables us to uncover similarities in the presence of various obfuscations. *SSDEEP* chunks the input sequence depending on the total sequence length, abstracts each chunk to a single character, and concatenates all characters to a hash. It then repeats this process with a doubled block size. Consequently, the resulting signature consists of the two hashes and the used block size. Using this technique, *SSDEEP* increases the abstraction level of the hash by the sequence length of the input.

Table 3.2 shows an example method `compare(int, int)` displayed in the first four representations. We excluded the fifth representation since it represents a fuzzy hash of the previous representation SPR. First, the AR removes the offsets and replaces the jump addresses of the offsets 2, 6, 11, and 15. Last, the SPR removes all type annotations of the instructions and transforms the jump instructions to `if`-keys.

Table 3.4 shows all obfuscation techniques handled by the five representations and our approach *LibDetect*. Bytecode addresses no obfuscation technique, but our approach uses it to identify unchanged library code. The AR handles small structural changes and modifier changes. Each following representation handles more obfuscation techniques. However, *Code Restructuring* can only be handled with the combination with *LibDetect*. None of the representations or *LibDetect* can handle hidden functionality since, for this, an approach needs to execute an app, and *LibDetect* only performs a static analysis.

To improve the precision of our matching procedure, we extract the fully-qualified name of a method, its instruction count, its enclosing class, and its defining package as metadata.

3. Library Detection

Table 3.4.: Comparison of obfuscation to handling entities.

Obfuscation	BC	AR	NR	SPR	Fuzzy SPR	LibDetect
Name Mangling	✗	✗	✓	✓	✓	✓
Modifier Changes	✗	✓	✓	✓	✓	✓
Structural Changes to a Method's Implementation	✗	✓	✓	✓	✓	✓
Code Slicing	✗	✗	✗	✗	✓	✓
Code Restructuring	✗	✗	✗	✗	✗	✓
Method Parameters Manipulation	✗	✗	✗	✓	✓	✓
Constant Computation	✗	✗	✗	✓	✓	✓
Fake Types	✗	✗	✓	✓	✓	✓
Code Optimization	✗	✗	✗	✓	✓	✓
Hide Functionality	✗	✗	✗	✗	✗	✗

3.2.2. Lookup

In order to find library methods to which an APK's method might correspond, we need a database of known library methods that enables an efficient lookup. For the lookup, we hash all five abstract representations of known library methods using the SHA-1 function [ErJ01] (the two fuzzy hashes of the SPR are processed individually) and build an index for the hashes of each representation. For the lookup, we use these indexes to point to the methods' metadata.

Algorithm 1: Best-matching methods

Data: Method m

Result: Matching methods

```

1 matches  $\leftarrow$  lookup( $m.FQN$ ,  $m.BC$ );
2 if matches  $\neq \emptyset$  then return matches;
3 matches  $\leftarrow$  lookup( $m.FQN$ ,  $m.AR$ );
4 if matches  $\neq \emptyset$  then return matches;
5 for repr in  $m.\{AR, NR, SPR, FuzzySPR_1, FuzzySPR_2\}$  do
6   matches  $\leftarrow$  lookup(repr);
7   if matches  $\neq \emptyset$  then return matches;
8 return  $\emptyset$ ;
```

Given an APK method, we use Algorithm 1 to lookup potentially-matching library methods in our reference database. The algorithm looks for matches using our abstract representations in increasing order of abstraction. The first two lookups search the database for methods with a matching fully-qualified name (FQN) and the same BC or AR as the APK method. These two representations allow us to precisely identify library methods that are not or only slightly obfuscated. All subsequent lookups ignore the declaring classes' FQNs, to address *Name Mangling*. To ensure that we have not

missed a method, we perform another lookup with the AR and proceed with the other representations until we find at least one match or otherwise declare the method as non-library code. This way, we find potential matches even in the presence of strong obfuscation with the highest match confidence. The matches of weak obfuscated methods avoid unnecessary large sets of method matches on more abstract representations.

3.2.3. Method/Class Matcher

In the last step, we aggregate the APK methods for which we found potentially-matching library methods to library classes.

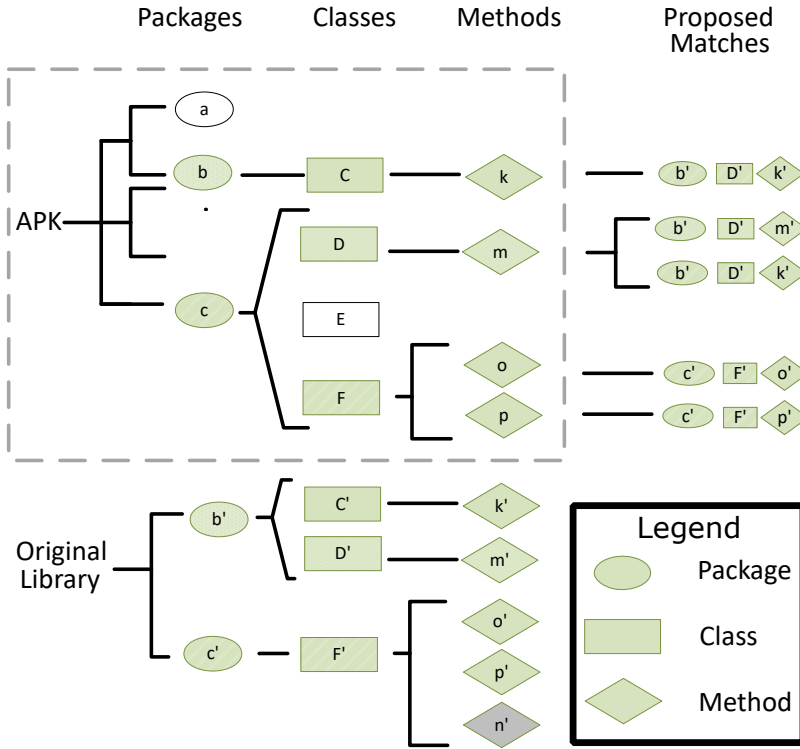


Figure 3.2.: Aggregating potentially-matching methods to library classes.

Figure 3.2 shows the app's structure, i.e., the packages (ellipse), classes (rectangles), and methods (diamonds) it contains. The code of each included library corresponds to a fragment of this structure (b, c, C, D, F, and k, m, o, p). Due to *Code Restructuring*, this fragment of a library may have a different structure than its respective original. Also, due to *Code Slicing*, elements of the original library might be missing (n' diamond shape).

For the aggregation of the code in Figure 3.2, *LibDetect* filters all app packages that match with code from the same library package and sorts them by the number of match methods from this library package. We refer to this number as the app-to-library-method count.

3. Library Detection

Afterward, *LibDetect* continues for each library package in descending order of the app-to-library-method count. For example, from app package *c* in Figure 3.2, two app methods *o*, *p* match methods from the library package *c'* and one app method match methods from the library package *b'*. Therefore, since more methods match from package *c* than from package *b*, *LibDetect* first processes *c* and then *b*.

Next, *LibDetect* computes for each library package a mapping from app classes to library classes. For this mapping, it considers only potentially-matching library methods from that library package. It maps each app class with the library class to which the highest number of methods match. If multiple classes match equally many methods, the procedure picks the library class with the closest size in terms of bytecode instructions. Each library class is mapped only once. For example, the app class *F* in Figure 3.2 is mapped to the library class *F'*, because two of its methods potentially match methods from *F'*.

Finally, *LibDetect* filters out class mappings that have less than half as many bytecode instructions as the mapped library class and reports the remaining mappings as library classes. The filtering of the described app classes avoids false positives caused due to a few methods that occur very frequently, especially in our abstract representations. For instance, if *LibDetect* matches only the initializers of a class without matching more than half of the code, then this class is removed from the mapping.

3.3. Evaluation

In this section, we evaluate *LibDetect*'s effectiveness in detecting library classes by answering the following research questions:

RQ1 How robust are our code representations against state-of-the-art obfuscation?

RQ2 How effective is *LibDetect* compared to other library detection approaches?

For the experiments we used a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The analyses were executed using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory and MySQL 5.7 for the library database.

3.3.1. Robustness of Code Representation

We assess the robustness of our code representations against obfuscation by applying *LibDetect* to obfuscated apps for which we know the used set of libraries.

Setup We downloaded from Maven Central [Mav17] all available 193 APKs for which the build files (POM file) document the used libraries. Afterward, we obfuscated these APKs using *DexGuard*, an extension of *ProGuard* [Pro17] that is integrated into the Android development environment and recommended by the Android developer board [pro20a]. Compared to *ProGuard*, *DexGuard* adds more advanced obfuscation techniques, in particular, string obfuscation and fake types. We use *DexGuard* with four

preset configurations: *Renaming*, *Optimization*, *String Obfuscation*, and all of them *Combined*. *DexGuard*’s default configuration enforces *Renaming*, therefore it is always enabled.

During the obfuscation process, *DexGuard* generates a detailed mapping file that specifies the origin of every obfuscated method. Using this information, along with the description of libraries stored in the APK’s POM file, we can assess whether *LibDetect* classifies a method correctly as belonging to a library.

Evaluation We filtered all methods ($\approx 26\%$) with less than ten instructions (e.g., simple getters and setters or default constructors), because, after name mangling, such methods are indistinguishable. Subsequently, we checked for each method, whether it is a library method and at which abstraction level *LibDetect* classified it as a library method (cf. Algorithm 1).

Results Figure 3.3 shows how often *LibDetect*’s method matching correctly classified a method as belonging to a library. For each configuration of *DexGuard*, the figure visualizes the percentage of methods found to the total number of methods per project using standard box-plots.

Observation 1 *Our results indicate that our representations are very robust against state-of-the-art obfuscation techniques (RQ1). In the case of Name Mangling, we correctly classified more than 95% of all methods. Moreover, if all obfuscation techniques are combined, we identified still more than 70% of all methods correctly.*

Figure 3.4 shows the importance of each representation. It depicts at which abstraction level/representation an obfuscated library method was detected. As expected, BC and AR could not find any library methods. Both rely on names, which are changed by the applied obfuscations in all configurations. However, these representations are still valuable in detecting unobfuscated methods that we have tested with the same set without applying obfuscation.

Interestingly, even if all obfuscation techniques are combined, the nameless representation (NR) already enables us to correctly classify a method in the vast majority of cases ($> 95\%$); this makes NR the most relevant representation. Nevertheless, SPR and fuzzy SPR are still needed to identify obfuscated library methods, and they are needed in more and more cases, increasing by 0.5% per obfuscation technique. Overall, the results indicate that our design decision to consider the representations in increasing order of their level of abstraction is helpful.

3.3.2. Library Detection

In this section, we present the results of the comparison between *LibDetect* and other library detection approaches.

3. Library Detection

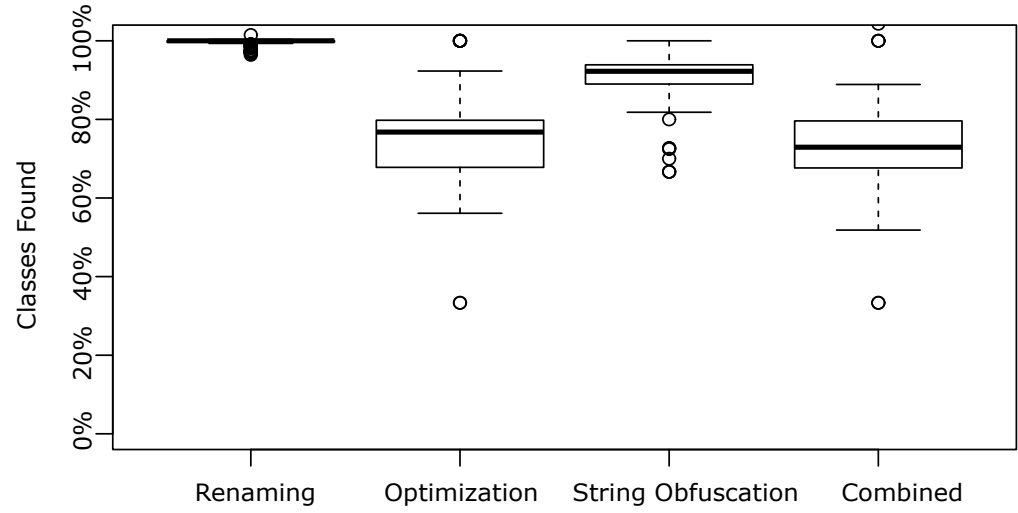


Figure 3.3.: Detection rates for different *DexGuard* configurations

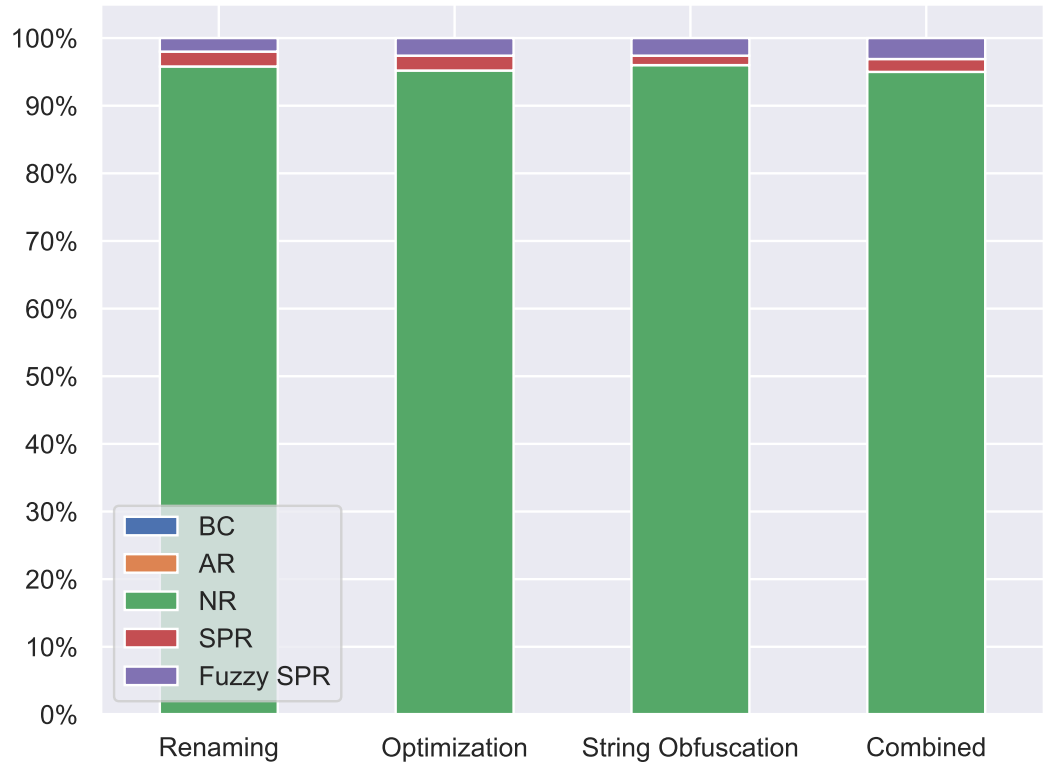


Figure 3.4.: Relevance of representations

Setup We collected 8,000 Android related libraries: $\approx 7,000$ from Maven Central and $\approx 1,000$ additional JARs collected manually by searching for package names from the

common-library list of Li Li et al. [LKLT⁺16] and the package names of *LibRadar*’s database [MWGC16] (cf. short description of *LibRadar* in 3.1). The Maven Central JARs were collected by analyzing the latest versions of the build files with dependencies to Android APIs (keyword “android” in the group ids of the dependency). Using all 8,000 libraries, we build the reference database, as described in Section 3.2.

For the evaluation of *LibDetect* on apps in the wild, we randomly selected 1,000 apps (99% confidence level; 5% confidence interval) from the app stores Anzhi [Anz17], Google Play [pla17a], App China [App17b], HiApk [HiA17], and Freewarelovers [Fre17]. Afterward, we measured the precision and recall of *LibDetect*, the common library white list (*Common Libraries*) by Li Li et al. [LKLT⁺16], and *LibRadar* [MWGC16]. We chose the five app stores to avoid biases, such as only small apps or only language-dependent apps. To determine the ground truth, we first identified the libraries used by the apps through manual code inspection, which enabled us to assess both the precision and recall of all approaches.

Results Figure 3.5 shows the average precision and recall of each approach. The *Common Libraries* set has an average precision of 99.6 % and a recall close to 9.3 %. *LibRadar* has an average precision of 99.7 % and recall of 11.5 %. *LibDetect* has an average precision of 80.2%, and an average recall of 87.2%.

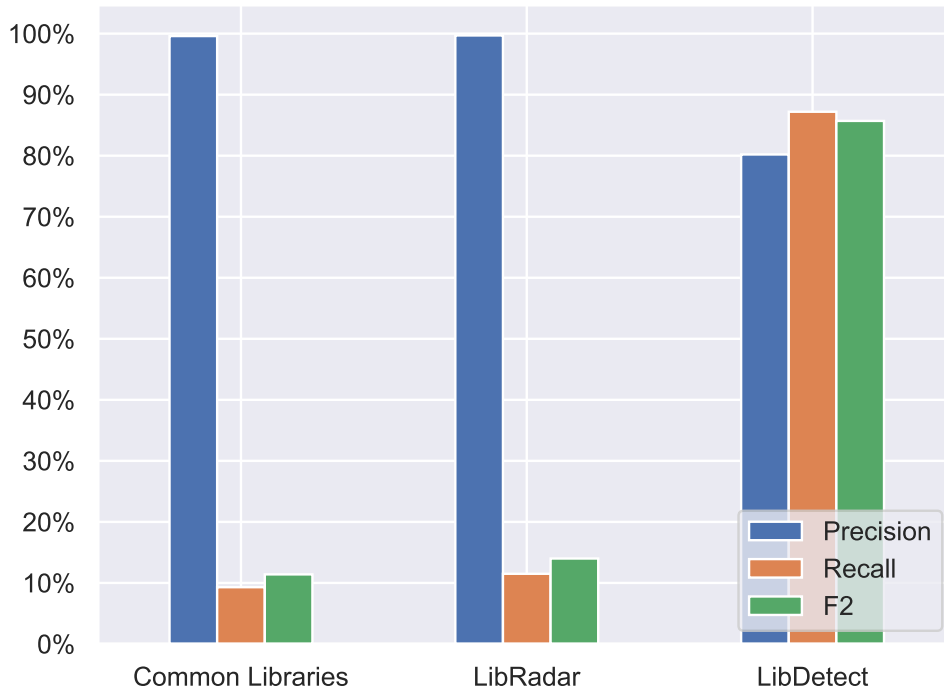


Figure 3.5.: Average precision, recall and F2-measure of the different library detection approaches

3. Library Detection

Discussion Given that library code causes false positives in code-based security analysis, identifying as many library classes as possible (i.e., a high recall) is essential. In order to reflect this statement, we calculated for Figure 3.5 the harmonic-balanced F_2 -measure, which weights recall higher than precision.

Observation 2 *LibDetect is more effective than the state-of-the-art approaches (RQ2). In our experiments, it identified the majority of library code (F_2 -measure of 85.7%) and outperformed the white list with Common Libraries and the tool LibRadar.*

LibRadar's low recall 11.5% is due to its package-level abstraction. As discussed above, the approach misses library classes that obfuscators move across package boundaries.

A careful analysis of the false positives/negatives of *LibDetect* revealed that the false positives are primarily due to **Activity** and **Listener**-classes that often occur in UI-intensive apps and which can be found in app-code as well as in library code. These classes often have very similar functionality and differ only in their names. Hence, they are indistinguishable to *LibDetect* because the (original) names are no longer available. The filtering of potentially obfuscated data-container classes of libraries caused *LibDetect*'s false negatives; in general, a container class primarily declares several fields along with respective getters and setters and, as discussed earlier, such short methods are filtered out.

3.3.3. Summary

LibDetect identifies library versions based on identified library classes, but does not map library classes. However, security analyses can use *LibDetect* to filter out library classes to focus solely on the app code.

The robustness experiment shows that *LibDetect* is suitable against many obfuscation techniques, especially *Code Restructurings* and *String Obfuscation* render many library detection approaches unusable. While *LibDetect* does not share many drawbacks of other approaches, it is not suitable for the detection of small library classes because these classes are too similar to other classes that belong to the app code. Therefore, we omit small classes from our analysis.

The authors of *Orlis* compared it against *LibDetect* and showed that *Orlis* has higher precision and recall for library detection of classes. However, we designed *LibDetect*, not for library mapping of classes but library class detection. As the authors of *Orlis* confirm, the precision and recall of *LibDetect* for mapping libraries based on sub-packages are significantly higher than those for classes. Besides the higher mapping ability of *Orlis*, it is vulnerable to *Code Restructuring*, *Method Parameter Manipulation*, *String Obfuscation*, and *Code Slicing*.

3.4. Threats to Validity

In the following, we discuss the threats to the validity that are related to our library detection experiments.

We construct *LibDetect*'s reference database from all libraries in *LibRadar*'s database, the latest Android related libraries from Maven Central, all libraries from a public white-lists [LKLT⁺16] and a popular-libraries list [mus17]. Nevertheless, the database may not contain all libraries used by apps in our evaluation datasets. In this case, *LibDetect* may fail to identify some of the library classes, and cause false positives or false negatives, but – given the size and quality of the data set – the overall error should be negligible.

Assembling an extensive, up-to-date reference database for *LibDetect* might be unpractical. We argue that since we include only public libraries listed in public databases and lists, assembling the database can be fully automated, which would allow frequent updates without any manual effort.

The evaluated apps from Maven Central may not be representative of apps in general. We chose these apps because they document their library dependencies, which allowed us to construct a ground truth for evaluating the library detection tools (see Section 3.3.2). The experiment shows how well *LibDetect* can discover (library) code embedded in a completely-obfuscated app. The impact of specific apps on the results of these experiments should be rather small.

In order to evaluate the impact of obfuscation on library detection, we used the obfuscator *DexGuard*, which applies renaming, optimization, shrinking, and string obfuscation. To the best of our knowledge, no existing library detection handles more advanced techniques. Other obfuscators might apply different or stronger techniques, such as virtualization, class encryption, class loading, control-flow flattening, and packaging. To the best of our knowledge, no existing library detection approach handles the latter techniques.

3.5. Conclusion

In this chapter, we presented a library detection approach that uses five representations to identify obfuscated libraries as precise as possible with high recall. For the detection, our approach *LibDetect* uses the matched library methods to aggregate them to entire library classes and evade multiple obfuscation techniques.

In our evaluation, we showed the robustness of our representations and the effectiveness of *LibDetect*. Our experimental setup demonstrated that *LibDetect* outperforms state-of-the-art library detection approaches by identifying more than six times more library classes. Furthermore, our work inspired the development of another approach *Orlis* [WWZR18] that improved the mapping of library classes.

4. App-Code Separation

Many approaches analyzed Android apps to identify whether developers obfuscated their code [WHA⁺18, Don18], the app contains malware [FGL⁺13, FLB⁺15, AMSS15, ARSC16, STDA⁺17], or the apps are repackaged [YFM⁺17, IWAM17, STDA⁺17, and17, LLB⁺17]. These approaches separate library code from the main app code to reduce the influence of library code on their analyses.

Currently, some approaches use library detection to separate app code. While library detectors such as *LibDetect* require a database of all library methods from all versions of each library to identify the library correctly, maintaining such a database to separate app code requires additional effort. For instance, some libraries are only released in an obfuscated form (cf. Section 7.3), which makes the identification of all associated methods challenging. Furthermore, the comparison of app methods against library methods generates additional overhead during the analysis. This overhead increases with each added library because the comparison algorithm needs to check the method signature for each new library. The additional overhead slows down analyses that use library detection as a prefilter.

In order to reduce the additional overhead caused by library detection, we propose *AppSeparator* that splits application code from the included library code without using a database with method representations that need to be maintained for each version of a library class. We based the implementation of *AppSeparator* on the separate compilation assumption [AL12], which states that even though the app code has dependencies on library code, this is not true vice versa. This assumption stems from the fact that libraries are independent modules that developers integrate into their apps. As a consequence, libraries have no explicit dependencies on a specific app.

Other approaches [LVHBCP14, LWW⁺17, STA⁺16] tried to build an app separation based on this assumption, but have not considered name obfuscation or multiple packages for app code. For instance, one approach uses only all classes contained in the main package defined in the Android manifest [LVHBCP14]. However, a developer can insert additional packages that contain parts of the app code. Additionally, no previous approach can identify obfuscated classes that were moved into other packages because these approaches focus solely on packages that are changed using *Name Mangling*.

To prepare the separation of app code, *AppSeparator* extracts all entry points of an app and all classes that transitively access these entry points. All these classes are flagged as potential app code. However, since some entry points may belong to library code, *AppSeparator* uses a white list [LKLT⁺16] to filter out these entry points. We use a white list because we assume that entry points are hard to obfuscate [WLG⁺17], as mentioned in Section 2.2. After filtering the library entry points, *AppSeparator* uses a classifier to determine whether a class actually belongs to the app code since optimizations and

4. App-Code Separation

restructuring of code can result in the insertion of references to app code into a library class. The classifier uses as features for each class the unique class counts of how many classes are used by the class to be analyzed and how many classes use this class, also called fan-in and fan-out. Using these counts, *AppSeparator* trains the classifier to distinguish between app and library classes. During *AppSeparator*'s construction, we used the knowledge from Section 2.2 to avoid any information that can be obfuscated by these techniques.

To evaluate *AppSeparator*, we used the data set collected in Section 3.3.2. We used 80% of the data for training and testing and 20% to measure the performance of *AppSeparator* against two of the state-of-the-art library detection approaches. To identify the most suitable classifier for our data, we trained multiple classifiers on our training data set and measured the correctly classified instances and the misclassification rate using 10-fold cross-validation. Given the best classifier for our data, we compared *LibDetect* and *LibRadar* with *AppSeparator* using the 20% of the collected data. *AppSeparator* identified more app classes than the other two approaches. Furthermore, we found that the additional overhead for extracting the fan-in and fan-out is needed to identify app classes more accurately.

In the following sections, we present the related work for our approach in Section 4.1, our approach *AppSeparator* in Section 4.2, the evaluation in Section 4.3, threats to the validity of our experiments in Section 4.4, and conclude our contributions in Section 4.5.

4.1. State-of-the-Art App-Code Separators

In this section, we describe the related work for separating app classes from library classes. The related approaches are similar to the ones from the library detection chapter because all tools that identify library packages or classes could also be used to separate these packages or classes from one another. Since we have already shown the different weaknesses of the approaches in the previous chapter, we only summarize these weaknesses in Table 4.1.

The approaches [CLZ14, ZZJN12] that separate app code using white lists are vulnerable to *Name Mangling*, *Code Restructuring*, and *Code Slicing* because any change to the names or the structure of the library code cannot be compensated with a static list.

Most other approaches are vulnerable to *Code Restructuring* because their fingerprints change when an obfuscator restructures large parts of the code. While *Ordol* [TLS17] is immune to such obfuscation techniques, it is sensitive to *Method Parameter Manipulation* because *Ordol* could produce the same fingerprints for methods that have different parameters.

Additionally, approaches such as *LibScout* [BBD16], *LibID* [ZBK19], *Orlis* [WWZR18], or the one from Feitcher et al. [FR19] are evaded by *Method Parameter Manipulation* because their fingerprint changes.

Another threat for three approaches [WWZR18, TLS17, FR19] is *String Obfuscation*. While these approaches do not use strings in their fingerprints, the fingerprints con-

Table 4.1.: Categories of app code separation approaches

Approach	Representation	Name Mangling	Code Restructuring	Code Slicing	Code Optimization	Modifier Changes	Method Parameter Manipulation	String Obfuscation
White Lists [CLZ14, ZZJN12]	Package Names	✗	✗	✗	✓	✓	✓	✓
<i>LibD</i> [LWW ⁺ 17]	Package Inclusion/Inheritance	✓	✗	✓	✓	✓	✓	✓
<i>LibRadar</i> [MWGC16]	API Calls	✓	✗	✗	✗	✓	✓	✓
<i>LibScout</i> [BBD16]	Merkle-tree-Hashes	✓	✗	✓	✓	✓	✗	✓
<i>LibPecker</i> [ZDZ ⁺ 18]	Code Entity Info.	✓	✗	✗	✓	✗	✓	✓
<i>Orlis</i> [WWZR18]	Transitive Method Calls	✓	✗	✗	✓	✓	✗	✗
Feichtner et al. [FR19]	AST Vectors	✓	✗	✗	✓	✓	✗	✗
<i>LibID</i> [ZBK19]	Basic Blocks	✓	✗	✗	✓	✗	✗	✓
<i>LibFinder</i> [CWC ⁺ 16]	Centroids	✓	✗	✓	✓	✓	✓	✓
<i>LibSift</i> [STA ⁺ 16]	Package Dependencies	✓	✗	✓	✓	✓	✓	✓
<i>Ordol</i> [TLS17]	Method N-Grams	✓	✓	✓	✓	✓	✗	✗
Han Hongmu et al. [HLT18]	Control-Flow Graph	✓	✗	✓	✓	✓	✓	✓

tain representations of all methods and method calls, which can be changed by this obfuscation technique.

AppSeparator does not suffer from the same drawbacks as the other approaches because it does not require a database of all method representations. These representations are essential to identify libraries but are not needed to separate app classes from others.

Recap The discussion above indicates the need for a better app code separation that is resilient against the above-described obfuscation techniques without using an extensive database that stores all library representations. Without such a database, the maintenance of its entries can be avoided, which removes the overhead to keep the representations of such approaches up-to-date. The maintenance of such a database is tedious because no central register administrates these libraries, and new apps may use libraries that are not known to the general public.

4.2. AppSeparator

In this section, we describe our approach *AppSeparator* that separates app code from library code. For the design of our approach, we used the separate compilation assumption [AL12], which states that app classes depend on library classes but not vice versa, and library classes also depend on other library classes. Following this assumption’s intuition, we extract all classes in the main package, all entry-point classes, and all classes that transitively access the entry point classes.

To identify all transitive accesses, *AppSeparator* starts with all classes defined in the main package and all entry points and adds in each iteration additional potential app classes by checking whether they access the current set. *AppSeparator* completes its iteration if it does not find additional accesses to the set of potential app classes.

To avoid entry points that could belong to libraries, *AppSeparator* uses a white list of library package names to filter them out. We assume that a white list is sufficient for filtering because, as mentioned in Section 2.2, entry points are hard to obfuscate.

Because obfuscation can influence the dependencies of classes and blur the lines between app classes and library classes, we extract two sets of dependency information for each class and train a classifier based on this information. This classifier decides for each class, whether it belongs to the app or library code. The first set of dependencies contains the classes used by a class under analysis (fan-out), and the second one consists of classes that use the class under analysis (fan-in).

Figure 4.1 shows the process of *AppSeparator* to split the app code from the library code. It extracts the fan-in and fan-out set of each class, computes the entry point information, and classifies each class to belong to the app or library code.

Next, we explain each step in detail.

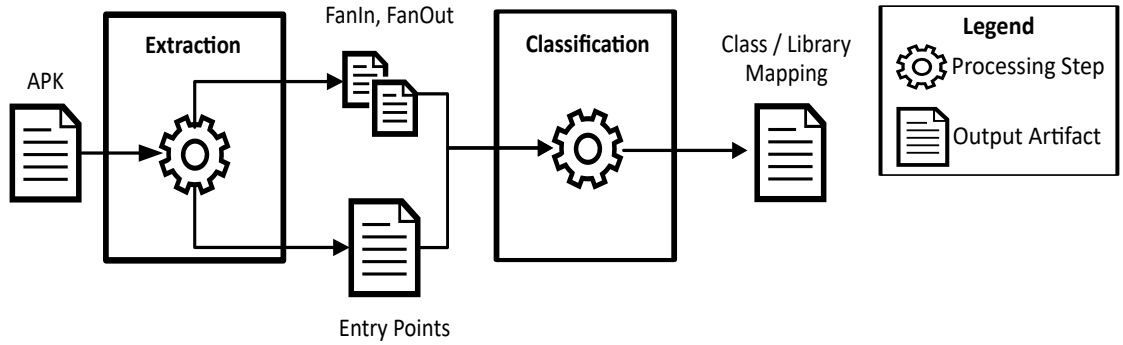


Figure 4.1.: Process of *AppSeparator* to classify classes into app or library code

4.2.1. Extraction

For the extraction of entry points, *AppSeparator* collects all subtypes of **Activity**, **Service**, **ContentProvider**, and **BroadcastReceiver** because these classes are the only ones that define entry points. In addition to the entry points, it extracts the app’s

namespace to identify the main app package. Using the entries and the namespace, we identify the initial set of classes that belong to the app code.

The entry points and the namespace are defined in the Android manifest. Consequently, we assume that obfuscators do not apply name mangling because they cannot keep track of all references of entry points [WLG⁺17]. Given this assumption, *AppSeparator* filters library entry points by using the first two segments of package names and comparing them with segments from a list of library package names (e.g., for the package name `com.google.android.test`, the first two segments would be `com.google`).

After the filtering, *AppSeparator* derives for each class its fan-in and fan-out set. To collect the fan-out set, *AppSeparator* extracts all field types, parameter types, return types of all methods, types of inheritance relations, and types used in the method bodies. Afterward, it excludes all base types or types that belong to the Android class library because, for the fan-out set, we consider only the relations between types that are bundled in the app's compiled code. Using the types of the fan-out set, *AppSeparator* deduces the fan-in set by collecting all reverse relations of the types. Given the app classes in packages of entry points and the fan-in/fan-out relations, we use Algorithm 2 to calculate the remaining potential app classes.

Algorithm 2: Determine potential app classes

Data: Initial App Classes `app_classes`, FanIn `fanIn`
Result: Set of potential app classes

```

1 queue ← app_classes;
2 result ← app_classes;
3 while queue ≠ ∅ do
4   current ← dequeue(queue);
5   if current ∈ fanIn then
6     for fi in fanIn[ current ] do
7       if fi ≠ current and fi ∉ result then
8         result = result ∪ {fi};
9         queue.enqueue(fi)
10      end
11    end
12  end
13 end
14 return result;
```

Algorithm 2 uses a work-list to iterate over all classes that use app classes and add them to the result set. In the beginning, the algorithm loads the initial potential app classes into a queue (line 1) and iterates over all class usages of the queued elements (line 6). If a class is already in the result set or equals the current class under analysis (line 7), the algorithm will continue with the next class from the current class's fan-in set. Otherwise, the algorithm inserts the class usage into the result set and the processing queue (lines 8 & 9) to analyze the usages of this class usage in another iteration. Once

4. App-Code Separation

the algorithm cannot add further classes to the queue, and all queued elements are processed, the algorithm returns all identified potential app classes. In the remainder of this chapter, we will refer to the extraction process as *IntuitionSeparator* because the resulting set of this algorithm can already be used to separate the app code of non-obfuscated apps.

4.2.2. Classification

Since the results of the *IntuitionSeparator* can be influenced by inserting false references or moving established references, we train a classifier that takes such actions into account. For the classification, we use the dependencies of all potential app classes as features to make our classifier immune to *Name Mangling*. Next, we describe the training and the used features, followed by the classification using the trained model.

Training We show in Table 4.2 all features that were gained from the information of the extraction process to train our classifiers.

Table 4.2.: Features for the REP Tree classifier

Feature	Description
Fan-in count	Number of unique class usages
Fan-out count	Number of classes that are used by the class
App fan-in count	Number of class usages that are potential app classes
App fan-out count	Number of potential app classes that are used by the class
Library fan-in count	Number of class usages that are potential library classes
Library fan-out count	Number of potential library classes that are used by the class
Is entry point	1 if the class is an entry point, 0 otherwise
Is app class	1 if the class is a potential non-library class, 0 otherwise

We chose these features based on the following intuitions and assumptions:

Fan-in/Fan-out We use the count of fan-in and fan-out to provide a classifier with an estimation for the coupling of the analyzed class. This measure’s intuition is that entry points of app code have a low fan-in count because other classes rarely access them. Consequently, classes that have high fan-out and a high fan-in count indicate no entry points of app code.

App fan-in/fan-out: The fan-in and fan-out of app classes indicate how many other app classes use the class under analysis or are used by it. While the usage of app classes stems from our overall intuition that app classes use library code but not vice versa, a high number of usages of the class under analysis by app classes can indicate a central app class such as a data container.

Library fan-in/fan-out: For these features, we assume that all classes that are not in the set of potential app classes belong to library code. The fan-in of these library

classes indicates that the class under analysis is also a library class. If the class under analysis uses many library classes, it can indicate a high coupling to these classes, which implies that the class under analysis is also a library class.

Is entry point: If the class under analysis is flagged as an entry point, and the class is not in a library package, this feature indicates an app class. We base this assumption on the statement in Section 2.2 that components and their corresponding references that reside in the Android manifest are hard to obfuscate.

Is app class: If the extraction process flagged the class under analysis as a potential app class, it indicates that the class belongs to the app code. A classifier gets a hint with this flag to compare the first assignment with other available features.

After extracting the above-described features, we train multiple classifiers to evaluate the most suitable classifier for our use case. The evaluation of the most suitable classifier is discussed in Section 4.3.2.

Classification After processing each class, the trained classifier returns a probability of whether a class belongs to the app code or library code. To get a binary decision, we use a threshold at the probability of 50%. We consider all values below 50% as library classes and all above or equal as app classes. If the classifier assigns a class to the app code, this class and all classes accessing this class will be flagged as app code for the next iteration to update the set of potential app classes for the next class under analysis. To ensure that different processing orders of classes do not influence the results of the classifier, we process the classes by the order of the app fan-out count. Given a classifier, we first perform the extraction and then the classification over all classes. In the end, *AppSeparator* returns a report that shows our assignment of classes to either library code or app code.

In the next section, we evaluate which classifier is the most suitable for *AppSeparator* and compare *AppSeparator*'s precision and recall with other approaches.

4.3. Evaluation

In this section, we evaluate the ability of *AppSeparator* to separate app code from the library code. In order to evaluate our approach, we investigated the following research questions:

RQ1: What is the best classifier for *AppSeparator* to split app code from library code?

RQ2: How accurate is *AppSeparator* compared to other library separators?

RQ3: What impact has the classifier of *AppSeparator* on its overall effectiveness?

RQ4: How effective is *AppSeparator* on obfuscated apps?

4. App-Code Separation

The following describes the data set used to train, test, and validate our approach. Then, we determine which classifier is the most suitable for our use case. Afterward, we compare *AppSeparator* with *LibDetect*, *LibRadar*, and *IntuitionSeparator* that uses the extraction of our approach without the classifier. In the end, we analyze *AppSeparator*'s effectiveness on different sets of obfuscated apps.

For our experiments, we used the following setup. The underlying server had two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The analyses were executed using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory.

4.3.1. Dataset

For our experiments, we built a ground truth to compare the results of our *AppSeparator*. In order to identify libraries in apps, we manually analyzed the code of the apps. As initial analysis, we used the common library list of Li Li et al. [LKLT⁺16] and the package names of *LibRadar*'s database [MWGC16] (cf. a short description of *LibRadar* in Section 3.1) as white list to identify names of library code in an app. Afterward, we analyzed all classes that are either obfuscated or not contained in the package names of the white list. Finally, each class that did not belong to the library code was assigned to the app code.

For the first three experiments, we selected 1,000 apps from five app stores (cf. Section 3.3.2) and randomly split the dataset into 800 apps for the training and test set and 200 apps for the validation set. We used the validation set to compare *AppSeparator* with other tools. Some of these apps contain obfuscated code, and some are not obfuscated. For this reason, we assessed, in the fourth experiment, *AppSeparator*'s ability to separate code in a controlled environment of obfuscated apps. Consequently, we collected all 453 non-obfuscated apps from the F-Droid store that contains open-source apps, which are not influenced by advanced obfuscation techniques. After downloading these apps, we obfuscated them using *ProGuard*'s standard configuration and a configuration to which we refer in further experiments as the advanced obfuscation configuration.

While the standard obfuscation configuration replaces meaningful names with meaningless random sequences of characters, the advanced configuration uses all additional options for the name obfuscation. This configuration enables the repackage option that removes all package names by moving all classes into the root package of the app and uses a natural-language dictionary for *ProGuard*'s name replacement procedure instead of using random character sequences.

4.3.2. Classifier Selection for AppSeparator

We analyzed Naive Bayes, Multilayer Perceptron, Logistic Regression, Random Tree, Random Forest, and REP Tree from Weka's collection [WF02] to assess the most suitable classifier for our data. While we trained most classifiers using their default values, we evaluated the Multilayer Perceptron using four different epoch values because of their impact on the classifier's performance.

For each classifier, we used Weka’s 10-fold cross-validation [HTF09] to train and test them on the 800 apps. Afterward, we compare the correctly-classified instances and root mean squared error of the classifiers given the cross-validation statistics. The root mean squared error is a measure for the false classifications of a model. During the 10-fold cross-validation, the trained classifiers processed a total of 1,171,616 classes.

Table 4.3.: Selection of classifiers for *AppSeparator*

Classifier	Correctly Classified	Root Mean Squared Error
Naive Bayes	80.57 %	37.83 %
Multilayer Perceptron (one epoch)	85.25 %	32.08 %
Multilayer Perceptron (five epochs)	85.82 %	31.65 %
Multilayer Perceptron (10 epochs)	85.89 %	31.55 %
Multilayer Perceptron (20 epochs)	85.92 %	31.47 %
Logistic Regression	84.65 %	34.46 %
Random Tree	87.19 %	30.37 %
Random Forest	87.22 %	30.13 %
REP Tree	90.21%	17.00%

Table 4.3 shows the correctly classified and false-classified measures of all classifiers. While the Multilayer Perceptron seems to improve after each epoch, the classification performance with more epochs only slightly increases. Both the Random Tree and Random Forest achieve more than 87% correctly-classified instances.

Observation 3 *We identified that the REP Tree yields the best scores with our training data. It correctly classified 90.21% instances and the false classification rate (root mean squared error) is only 17%. The results in Table 4.3 show that REP Tree is the most suitable classifier to process our data (RQ1), and therefore, we select it for AppSeparator.*

We assume that because the other decision trees are based on randomly selected features, they selected a hierarchy of features that resulted in poorer results. Furthermore, the other classifiers may produce unsatisfactory results because they cannot handle continuous values (Multilayer Perceptron) or assume conditions that do not hold (e.g., Naive Bayes assumes that all features are independent of each other).

4.3.3. Comparison against other Library Detectors

Given the validation set, we measured precision, recall, and F_1 —measure of *AppSeparator*, *LibRadar*, *LibDetect*, and *IntuitionSeparator* that uses only the extraction step of our approach without the classifier. By using *IntuitionSeparator* in our evaluation, we analyze the impact of the classifier on the performance of *AppSeparator*. We executed all approaches on the validation set consisting of 200-apps with 62,675 app classes and 246,758 library classes. Figure 4.2 shows the results of the four tools.

As we can see in Figure 4.2, *LibRadar* has the highest precision but at the cost of a poor recall of only 12.46%. While *LibDetect* has higher recall than *LibRadar*, it has a 20.31% lower precision.

4. App-Code Separation

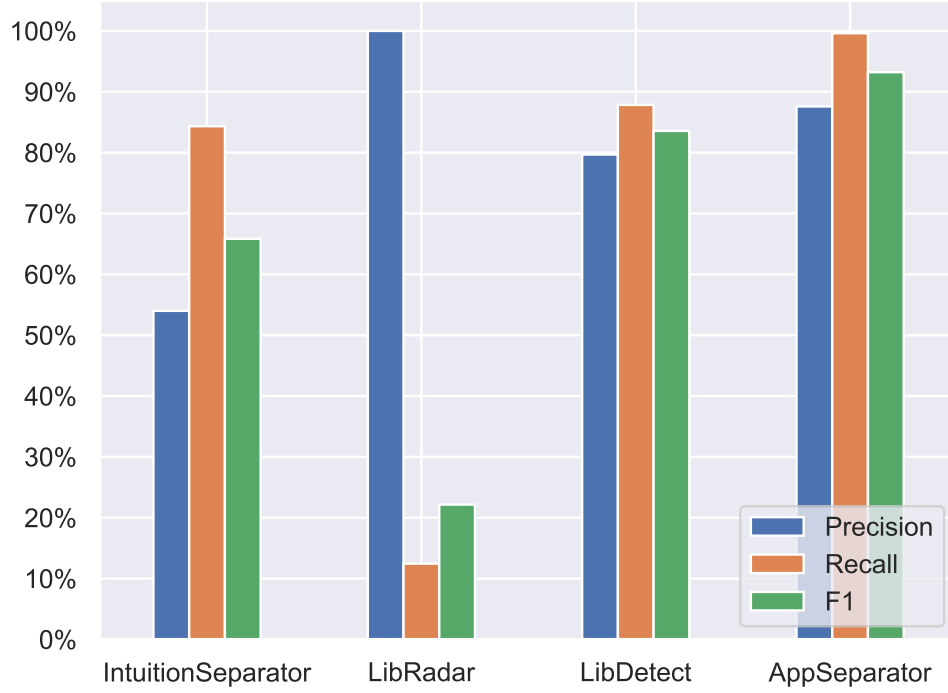


Figure 4.2.: Average precision, recall and F_1 -score of all approaches

Observation 4 Comparing *AppSeparator* with *IntuitionSeparator* shows that the classifier improves the detection performance significantly (**RQ3**). *AppSeparator*’s precision is about 38% times higher, and the recall is about 15% higher than the ones of the *IntuitionSeparator*.

Observation 5 *AppSeparator* is more accurate than all other analyzed tools; it has a recall of 99.62% and achieves a F_1 -measure that is at least 11.55% higher than the ones of the best-analyzed separator (**RQ2**). The precision of *AppSeparator* is the second highest, only surpassed by the precision of *LibRadar* that has the worst recall.

4.3.4. Performance on Obfuscated Apps

For this experiment, we used all 453 apps from the F-Droid store that were not obfuscated and obfuscated them using ProGuard’s standard and advanced obfuscation configuration. The F-Droid store is an open-source platform, so the analyzed code contains no complex obfuscation.

Because we obfuscated the apps, we acquired for each app the mapping files produced by *ProGuard*. These files contain which names the obfuscator mapped to which other

names. From these mapping files, we can deduce all library classes by using a common white-list [LKL⁺16] of library package names and comparing the package names with the original ones. Using these mapping files, we determine for each app whether *AppSeparator* separated the app classes from the library classes correctly.

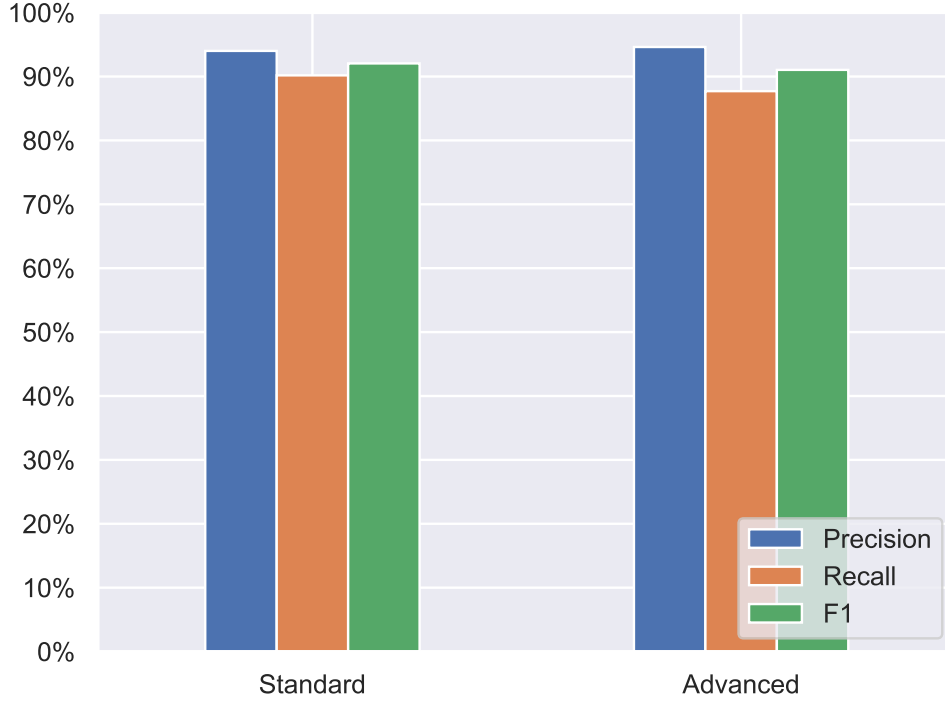


Figure 4.3.: Average precision, recall and F_1 –score of *AppSeparator* on both obfuscation configurations

Figure 4.3 shows the results of *AppSeparator* executed on the app set that we obfuscated using the two configurations. In the experiment with the standard configuration, *AppSeparator* has higher recall than in the one with the advanced configuration (90.18% vs. 87.71%). However, in the latter experiment, *AppSeparator* achieves a slightly higher precision than in the former (94.01% vs. 94.62%).

Observation 6 *The high recall and precision indicate that AppSeparator identifies library classes even when obfuscators change their name or restructure their code into other packages. Therefore, we can conclude that AppSeparator is very effective for app-code separation even for obfuscated apps (RQ4).*

The advanced configuration replaced all names belonging to library code and removed all reference points to separate the libraries. However, *AppSeparator* was still able to separate 87.71% of the classes correctly.

4. App-Code Separation

Observation 7 *Consequently, we claim that even though AppSeparator could not use its white list to filter out known library code, it separated most of the classes in an app.*

4.3.5. Summary

In the first experiment, we chose multiple classifiers to determine the most suitable among them to separate app classes from library classes. We chose these classifiers because they are commonly used in literature for similar problem domains and can process our input features.

In the second experiment, we compared *AppSeparator* with *IntuitionSeparator*, *LibRadar*, and *LibDetect*. We selected these separators because they are the only publicly available state-of-the-art approaches released with their complete database. *AppSeparator* outperformed the other approaches in all our experimental settings.

The third experiment shows *AppSeparator*'s performance on obfuscated apps by processing apps obfuscated with *ProGuard*'s standard name obfuscation and an advanced configuration for name obfuscation. While we used *ProGuard* to obfuscate the selected apps, other obfuscation tools use either the same name-obfuscation scheme or *ProGuard*'s name obfuscation as a post-processing step. *AppSeparator* correctly classified at least 87.71% of all analyzed library classes.

With these results, we conclude that *AppSeparator* is a powerful approach to separate app code from library code. It is even suitable if an obfuscator repackages all classes into a single package and manipulates all names of the app under analysis. We discuss threats to the validity of our evaluation in the next section.

4.4. Threats to Validity

We constructed *AppSeparator*'s reference white list using the two segments of the package names from all libraries in *LibRadar*'s database and a public white list [LKLT⁺16]. Nevertheless, both sources may not contain all library package names used by apps in our evaluation data sets. In this case, *AppSeparator* may have failed to separate some library code correctly, and a complete white list may lead to other results than the reported ones. However, as shown in the last experiment, *AppSeparator* has a significant impact on the overall result without relying on the white list, making the error rate of an incomplete white list negligible.

Assembling a ground-truth data set containing apps with all possible obfuscation scenarios and in all possible sizes is infeasible. However, our data set was obfuscated with the most used tool [WHA⁺18] for app obfuscation and in the most advanced configuration. Furthermore, most apps have a similar ratio between app code and library code [WGMC15], which makes the separation of app code for other apps similar to those in the learned data set.

The 200 evaluated apps from Section 4.3 may not be sufficient to compare the performance of *AppSeparator* with other tools. Since the other tools have a similar performance

on larger data sets, we assume that *AppSeparator* will also have a similar performance on the other data sets.

To evaluate the impact of obfuscation on app-code separation, we used the obfuscator *ProGuard*, which applies renaming, optimization, and shrinking. To the best of our knowledge, other obfuscators such as *DashO* [Das19], *DexGuard* [Dex19c], or *DexProtector* [dex19a] either use the same renaming technique as *ProGuard* or use it as a post-processing step. Both the post-processing and the similar technique lead to the same obfuscated names. Nevertheless, other obfuscators might apply stronger techniques, such as virtualization, class encryption, class loading, and packaging. To the best of our knowledge, no other existing approach can handle these techniques.

4.5. Conclusion

In this chapter, we presented *AppSeparator* that uses the single compilation assumption [AL12] to separate app code from library code. Following the intuition of this assumption, *AppSeparator* extracts all entry points of an app and all classes that transitively access the entries to identify app code. Since specific obfuscation techniques can blur the lines between library code and app code, *AppSeparator* additionally uses a classifier to assess whether a class belongs to the main app code. For the classification, *AppSeparator* uses the counts of different fan-in and fan-out sets of a class as features.

In order to evaluate *AppSeparator*, we compared its precision, recall and F_1 -measure with *LibRadar* and *LibDetect*. *AppSeparator* outperformed these tools in separating app code by a higher F_1 -measure of at least 11.55%. Furthermore, we showed that *AppSeparator*'s classifier is crucial for its performance and that even advanced name obfuscation does not heavily weaken the effectiveness of it.

5. Repackage Detection

Users install popular apps from the Google Play store on millions of devices [Ann20]. This wide-spread Android app usage attracts malicious actors to create altered, repackaged versions of those apps and steal the original owner’s revenue or trick users into infecting their mobile devices with malware. Detecting such repackaged apps is, therefore, necessary for a secure and viable app market.

Several techniques for repackaging detection have already been proposed and can be broadly categorized by their used representations:

Code-Statistics-based approaches [ZZG⁺13, LVHP16, ZGW⁺16, GHLZ16, SLQ⁺14, LLB⁺16] extract statistics about metadata, packages, and classes to identify similarities in repackaged apps.

View Graph-based techniques [ZHZ⁺14, CCS⁺15, SLL15, CWL⁺15, YFM⁺17, LLYZ16, STAW15] construct a graph by using views as nodes and the transitions from one view to another as edges.

Resource Hash-based approaches [ZGC⁺14, IWAM17, IWAM16, STDA⁺17, GLZ16] hash internal files of an app without considering the file content or type.

Data-, and Control-Flow Graph-based techniques [CGC12, CGC13, CLZ14, GSC⁺13, and17, ARSC16, LLB⁺17, HHW⁺12] derive the control-flow, or data-flow graph of the analyzed app and measure the similarity by comparing isomorphic sub-graphs of the derived properties. Given that graph matching is a hard problem, these approaches potentially suffer from scalability issues [CGC12].

Repackaging Anomaly-based approaches [FGL⁺13, FLB⁺15, GKS⁺15] identify repackaged apps by anomalies that are caused through the recompilation of an app.

High-Level Features-based approaches [CADZ15, PNNRZ12] try to decompile the code or extract other features that are mostly only available in non-obfuscated apps to identify repackaging. However, most repackagers obfuscate their apps so that decompilation is almost infeasible.

Android API-based approaches [SZX⁺14, ZSL13, KGGS16, WGMC15] extract features from API calls to construct a fingerprint that they use to identify repackaged apps.

Execution Trace-based approaches [WZSL15, LLCT13, AMSS15, MZW⁺16] execute apps in an emulated environment and extract essential traces of the execution. However, an obfuscator can inject guards into the code to evade such approaches.

Fuzzy-Hash-based approaches [ZZJN12, QPX⁺16] dissect the core functionality of an app, create signatures based on an apps’ code, and compare it with potential

5. Repackage Detection

repackaged apps using a fuzzy hash that is immune to small code changes. Our proposed approach also belongs to this category.

Challenges A challenge for all existing repackaging-detection techniques is code transformations. Developers regularly use *Name Mangling* and *Code Optimization* to increase the performance of their apps. Additionally, they obfuscate their apps to protect their intellectual property. However, attackers also apply obfuscation to hide malicious code and evade signature-based detectors, such as anti-virus software.

Current repackaging-detection techniques can only handle rudimentary forms of obfuscation, such as one-by-one identifier renaming, replacing types, and reordering fields and methods [MWGC16, BBD16]. More sophisticated obfuscation techniques, such as moving classes between packages or changing Android API-calls, are not supported. Our evaluation of Google Play store apps revealed that 60% [rev17] are at least partially obfuscated, and 20% of them repackage their classes into the root package. The prevalent reuse of libraries in apps further inhibits the effectiveness of repackaging detection. Wang et al. [WGMC15] reported that more than 60% of the sub-packages in Android apps belong to library code. Hence, separating the library code from the app code is necessary. Otherwise, apps that use (nearly) the same libraries automatically share a large portion of the overall codebase. Thus, detectors always identify them as repackaged – even if the apps’ code is entirely different. Hence, libraries used in an app need to be identified and removed before measuring the similarity to other apps. A basic approach to filter out *non-obfuscated library code* is to use package white-lists [CLZ14, ZZJN12]. However, these approaches do not handle obfuscated package names or the relocation of app classes into library packages.

Another challenge for repackaging-detection tools are apps generated by App Makers, e.g., apps-builder [app17a]. Using such makers, the vast majority of the codebase – the generator’s libraries – will be the same, and the rest will still be very similar. Current approaches will generally flag such apps as repackaged. Such apps are generated by App Makers (E.g., apps-builder [app17a]), which enable users to generate apps without coding knowledge. Previous repackaging-detection tools entirely overlooked this kind of apps and flagged them as repackaged.

In general, the detection of the used libraries is not easy because the compiler fuses the app’s codebase with the libraries’ codebases, i.e., detectors cannot find the libraries in some archive file or a particular folder, and they cannot identify the libraries using some meta-information. Additionally, it is a recommended and applied practice to obfuscate the entire or at least some parts of the codebase, which renders approaches based on package names infeasible.

Proposed Approach To address the identified challenges, we propose *CodeMatch* that detects repackaged apps using our library-detection technique *LibDetect* (cf. Chapter 3) and our app code separator *AppSeparator* (cf. Chapter 4).

CodeMatch uses fuzzy hashing [Kor06] of an app’s code to withstand various sophisticated obfuscation techniques and optimizations, including; class relocating, slicing,

duplication of Android APIs in the app, code changes, and code optimizations that affect the detection. Additionally, it orders the app’s packages based on their classes’ size, which addresses the challenges faced by *DroidMOSS* [ZZJN12] due to reorderings of classes and packages.

CodeMatch performs the following steps to identify repackaged apps:

- First, it filters apps that are generated by tools (App Makers). These generated apps account for 2.4% of all analyzed apps, and they generally share the vast majority of their codebase. We can filter them out using a white list of the main-package prefixes. Since the Android signing process requires these prefixes, obfuscators cannot manipulate them.
- Second, *CodeMatch* filters the library code of apps using *LibDetect* and *AppSeparator*.
- Third, it filters apps with less than ≈ 300 lines of code; our approach cannot reliably classify such apps through the size limitation of fuzzy hashing.
- Fourth, *CodeMatch* generates for each app the most abstract/obfuscation-resilient representation from Section 3.2.1 and fuzzy hashes it.
- Fifth, it compares the fuzzy hashes with the ones of other apps, and if the similarity exceeds a predefined threshold, the apps are marked as repackaged.

We prepared the evaluation of *CodeMatch* by fuzzy hashing the descriptions of downloaded apps and randomly selecting 1,000 app pairs, whose fuzzy-hashed descriptions are at least 90% similar; we considered very similar descriptions as the first indicator for repackaging. Afterward, we installed and executed each app pair to reconfirm their similarity manually. We used these results as the ground truth, to evaluate the findings of *CodeMatch*, *ViewDroid* [ZHZ⁺14], *DroidMOSS* [ZZJN12], *FSquaDra* [ZGC⁺14], and an repackaging detection that uses the centroid concept from physics [CLZ14]. We selected these tools because they are publicly available, or we could re-implement them based on their documentation.

Additionally, we evaluated the effects of *CodeMatch* with different library separators. To evaluate the effect of *CodeMatch* in isolation, we additionally run it with a library white-list, with *LibRadar*, and *AppSeparator* (cf. Chapter 4). We show that *CodeMatch* enables us to identify, in all library-detection configurations, up to 50% more obfuscated and repackaged apps than the other approaches. In summary, we make the following contributions:

- *CodeMatch*, a technique that detects app repackaging, which uses *LibDetect* and *AppSeparator* to filter out libraries before measuring the similarity of the apps.
- A quantitative comparative evaluation of available repackaging detection approaches (*CodeMatch*, *ViewDroid*, *DroidMOSS*, *FSquaDra* and a *centroid-based* approach)

5. Repackage Detection

- A list of package names that generally identify applications built using App Makers, e.g., <http://www.apps-builder.com>, and therefore share the entire codebase, but are generally not repackaged [rev17]

The remainder of this Chapter is structured as follows. Section 5.1 presents the attacker model. Section 5.2 describes the state-of-the art. Section 5.3 presents the proposed approach. Section 5.4 discusses the results of our evaluation. Section 5.5 examines threats to the validity of our experiments. Finally, Section 5.6 concludes the contributions of this chapter.

5.1. Attacker Model

We identified three kinds of attackers who create repackaged apps, which we categorize based on their knowledge and manual effort to trick repackaging detectors.

Basic Knowledge: Attackers from the first category only apply basic changes/obfuscations to an app, which do not require a deep understanding and configuration of obfuscators. Their primary goal is to avoid the detection of their repackaged app by hash-based approaches.

More Knowledgeable: The second category makes full use of existing advanced obfuscators to hide their apps effectively, even if analysts use state-of-the-art repackaging detection approaches. They use advanced obfuscators that are automatically able to add arbitrary code and circumvent more advanced repackaging detectors. These attackers are dangerous because they can automatically repackage apps. Hence, they can flood an entire app store with repackaged apps.

Expert: The third category of attackers are experts who can apply custom obfuscation techniques. Their obfuscation repertoire for repackaged apps includes, but is not limited to, manual code changes, using self-written obfuscators, or applying compression or encryption to a part of the app. While the repackaged apps of these attackers are hard to identify, their focus lies on a small number of specific apps because of the vast amount of manual effort.

Current repackaging detection tools can identify repackaged apps created by attackers from the first category. Our proposed approach – *CodeMatch* – is additionally able to identify repackaged apps of attackers from category two.

5.2. State of the Art

This section presents the state-of-the-art approaches for repackaging detection of Android applications. During the analysis of related work, we identified 40 different approaches [LBK19] that either handle repackaged apps directly or in order to identify malicious content. Table 5.1 shows the identified approach sorted by their library filtering, their internal representation, and potential weaknesses against the obfuscation

techniques described in Section 2.2. For brevity, we shorten *Name Mangling* to *NM*, and *Code Restructuring* to *CR*.

Since the detailed description of all 40 approaches would go beyond the scope of usefulness, we describe approaches grouped according to their representations and limitations in terms of obfuscation techniques. In Table 5.1, we divided the groups using horizontal lines, and we will describe each group in the following from top to bottom of the table.

Metadata-, Package-, and Class Statistics Approaches [ZZG⁺13, LVHP16, ZGW⁺16, GHLZ16, SLQ⁺14, LLB⁺16] from this group extract package, class, or metadata statistics to create a fingerprint for an app and compare it with ones extracted from potential repackaged apps.

Unfortunately, in this category, no approach filters library classes that can cause high similarity scores among apps that use the same or a similar set of libraries. Additionally, repackagers can use *Code Restructuring* to manipulate the metadata-, package-, or class information to evade the detection of repackaged apps.

Fuzzy Hashes *DroidMOSS* [ZZJN12] and *T-CTPH* [QPX⁺16] use a white-list to filter out libraries and create from an app’s opcodes a fingerprint by fuzzy hashing the entire opcode sequence in the given order. Afterward, both approaches compare the created fingerprints against potentially repackaged apps.

Although the approaches use fuzzy hashing to identify repackaging in obfuscated apps, they are vulnerable to *Code Restructurings* because the fingerprints depend on the code order of an app. Additionally, both approaches suffer from *Name Mangling* because the processing order of each fuzzy hash depends on names, and the order can change by manipulating the names of packages, classes, methods, and fields.

View Graphs Approaches [ZHZ⁺14, CCS⁺15, SLL15, CWL⁺15, YFM⁺17, LLYZ16, STAW15] from this group extract view graphs based on activities as nodes and their actions as edges, e.g., button pushes or intent executions. These approaches compare the view graphs of an app with potential repackaged ones. For instance, *ViewDroid* [ZHZ⁺14] extracts a viewgraph and compares the viewgraphs of an app with potential repackaged ones by measuring their sub-graph isomorphisms.

Not only *ViewDroid* perform sub-graph similarity measurements for the comparison, but all these tools. However, the comparison with such a technique is very time-consuming. Additionally, all approaches are vulnerable to the insertion of libraries with their views because they do not filter libraries. Furthermore, most approaches can only compare apps with more than three views. In a random sample of 1,000 apps, 44.3% had less than 3 views.

Resource Hashes These approaches [ZGC⁺14, IWAM17, IWAM16, STDA⁺17, GLZ16] either extract hash values for all files referenced in the `MANIFEST.MF` or compute their hash values from various code elements and resource files. Afterward, the approaches compare these hash values with the ones extracted from potentially repackaged apps.

5. Repackage Detection

Table 5.1.: Categories of Repackaging Detection Approaches

Approach	Library Detection	Representation	Obfuscation Weakness
<i>PiggyApp</i> [ZZG ⁺ 13]	✗	Package Fingerprint	CR
<i>CLANdroid</i> [LVHP16]	✗	Cluster of Meta Data	CR
Jingqiu Zheng et al. [ZGW ⁺ 16]	✗	Class Relations	CR
<i>RepDetector</i> [GHLZ16]	✗	Class Statistics	NM & CR
<i>Resdroid</i> [SLQ ⁺ 14]	✗	Main Package Structures	CR
Li Li et al. [LLB ⁺ 16]	✗	Package Statistics	NM & CR
<i>RomaDroid</i> [KLCP19]	✗	Manifest Structure	CR
<i>DroidMOSS</i> [ZZJN12]	White-List	Fuzzy Hashes	NM & CR
<i>T-CTPH</i> [QPX ⁺ 16]	White-List	Fuzzy Hashes	NM & CR
<i>ViewDroid</i> [ZHZ ⁺ 14]	✗	View Graph	✓
Cuixia et al. [CCS ⁺ 15]	✗	View Graph	✓
<i>DroidEagle</i> [SLL15]	✗	View Graph	✓
<i>MassVet</i> [CWL ⁺ 15]	✗	View Graph	CR
<i>RepDroid</i> [YFM ⁺ 17]	✗	View Graph	NM & CR
<i>SUIDroid</i> [LLYZ16]	✗	Hashes of View Graph	✓
Charlie Soh et al. [STAW15]	✗	Activity Flows	CR
<i>FSquaDra</i> [ZGC ⁺ 14]	✗	Resource Hashes	NM & CR
<i>APPraiser</i> [IWAM17, IWAM16]	✗	Resource Hashes	NM & CR
<i>DroidSieve</i> [STDA ⁺ 17]	✗	Resource Hashes	CR
Olga Gadyatskaya et al. [GLZ16]	✗	Resource Hashes	CR
<i>DNADroid</i> [CGC12]	✓	Data Dependency (DD)	NM & CR
<i>AnDarwin</i> [CGC13]	✓	Cluster of DD	CR
<i>Adrob</i> [GSC ⁺ 13]	✗	Cluster of DD	NM & CR
<i>AndroGuard</i> [and17]	✗	Control-Flow Graph	CR
<i>DroidClone</i> [ARSC16]	✗	Control Flow Graph	CR
Li Li et al. [LLB ⁺ 17]	✗	Control-Flow Graph	CR
<i>Juxtapp</i> [HHW ⁺ 12]	✗	Basic Block N-Grams	NM & CR
Kai Chen et al. [CLZ14]	White-List	Centroids	NM & CR
<i>AndroSimilar</i> [FGL ⁺ 13, FLB ⁺ 15]	✗	Improbable Features	NM & CR
<i>AndroidSOO</i> [GKS ⁺ 15]	✗	Recompilation Anomalies	CR
Jian Chen [CADZ15]	✗	Code Tokens	CR
Rahul Potharaju et al. [PNNRZ12]	✗	AST Symbols	CR
<i>DroidSim</i> [SZX ⁺ 14]	✗	API Graph	CR
<i>DroidAnalytics</i> [ZSL13]	✗	API Hashes	CR
Daeyoung Kim et al. [KGGS16]	✗	API N-Grams	CR
<i>Wukong</i> [WGMC15]	✓	Cluster of APIs	CR
Xueping Wu et al. [WZSL15]	✗	HTTP Traffic	CR
<i>SCSDroid</i> [LLCT13]	✗	System Call Sequence	CR
<i>PICARD</i> [AMSS15]	✗	System Call Graph	✓
<i>LoPD</i> [MZW ⁺ 16]	✗	Symbolic Execution Traces	CR

For instance, *FSquaDra* [ZGC⁺14] extracts the hash values for all files referenced in the `MANIFEST.MF` and compares them with the ones of a potentially repackaged app.

While these approaches are usable for near-to identical, repackaged apps, they are not suitable for obfuscated resource files. For instance, since none of the approaches has a library detection, the insertion of library code would change the entire hash value of the code elements. Even adding some (useless) resources (e.g., sound files or images) would change the computed hashes and reduce the similarity with original apps.

Data- and Control-Flow Dependencies Approaches [CGC12, CGC13, CLZ14, and17, GSC⁺13, ARSC16, LLB⁺17, HHW⁺12] that use data- or control-flow dependencies, extracts these dependencies from each method, and create a signature to match them against methods of potentially repackaged apps. For instance, the approach of Kai Chen et al. [CLZ14] uses control-flow graphs to calculate the centroids of each method and use these centroids to identify repackaged apps. Physicists use centroids to describe the perfect point to balance any given shape.

While all of the approaches are vulnerable against *Code Restructuring*, the similarity measurements of six approaches also might be highly biased towards libraries because they use either only a white-list or no library filtering.

Repackaging Anomalies Some approaches [FGL⁺13, FLB⁺15, GKS⁺15] attempt to identify features that hint at the recompilation or repackaging of an app.

Since some obfuscators recompile an app's code, the mentioned approaches might confuse obfuscated apps with repackaged ones. The recompilation of a repackaged app could also have the opposite effect on the detection. For instance, a repackaged app after obfuscation with *Name Mangling* or *Code Restructuring* could no longer possess the anomalies necessary for detection. Additionally, the build process of an app might recompile used libraries and influence the extracted features.

High-Level Features The approaches by *Jian Chen* [CADZ15] and Rahul Potharaju et al. [PNNRZ12] extract high-level features such as while-loops from the abstract syntax tree (AST) of the decompiled app. Afterward, they use these features to compare the original apps against potential repackaged ones. If the apps have a high similarity, the approaches flag the latter app as repackaged.

However, the obfuscation of Android apps makes the complete decompilation or extraction of AST from an app's code nearly impossible. Therefore, this approach would miss a large amount of repackaged apps. Additionally, both approaches suffer under false positives due to missing library detection.

API-Based Detection Several approaches [SZX⁺14, ZSL13, KGGS16, WGMC15] extract features based on API calls by building graphs, hashes, n-grams, or clusters from them. Afterward, the approaches use these features to detect repackaged apps.

While all approaches would suffer from *Code Restructuring*, not all have a high false-positive/negative rate in the detection results caused by integrated libraries. *Wukong*

5. Repackage Detection

filters libraries before constructing an API vector by using *LibRadar*. However, since *LibRadar* filters libraries based on the app’s package structures, it might miss libraries that obfuscator split by *Code Restructuring*. Therefore, if the filtering misses one of the libraries, *Wukong* has a high dissimilarity in its results so that repackaged apps remain undetected.

Execution Traces Approaches [WZSL15, LLCT13, AMSS15, MZW⁺16] from this group execute apps in an emulated environment and extract execution traces to build graphs, HTTP-traffic signatures, or other fingerprints. Afterward, the approaches compare these fingerprints against ones extracted from potential repackaged apps.

While these approaches can identify repackaged apps with the same execution traces, repackaged apps that contain guards against dynamic analyses or have more complex execution traces evade the capabilities of these approaches. Additionally, library code biases their similarity measurements because the missing filtering leads to a high similarity-value between execution traces of non-repackaged apps.

Discussion To recap, each repackaging detection approach has its specific drawbacks; all share problems due to limitations of the library filtering in use. To address these problems, we designed *CodeMatch*, which we evaluate against *DroidMOSS*, *ViewDroid*, *FSquaDra*, and the centroid-base approach in Section 5.4. For *FSquaDra* and *ViewDroid*, the software was either available online or was made available to use upon request. The code for *DroidMOSS* and for the *centroid*-based approach was not available; but we were able to re-implement them based on the information available in their publications [ZZJN12, CLZ14]. We could not acquire the remaining approaches from their authors, and the re-implementation of all approaches based on their publications is infeasible.

5.3. CodeMatch

Figure 5.1 depicts the workflow of *CodeMatch* that is separated in filtering, the building of the representation, and comparing it with representations from the database.

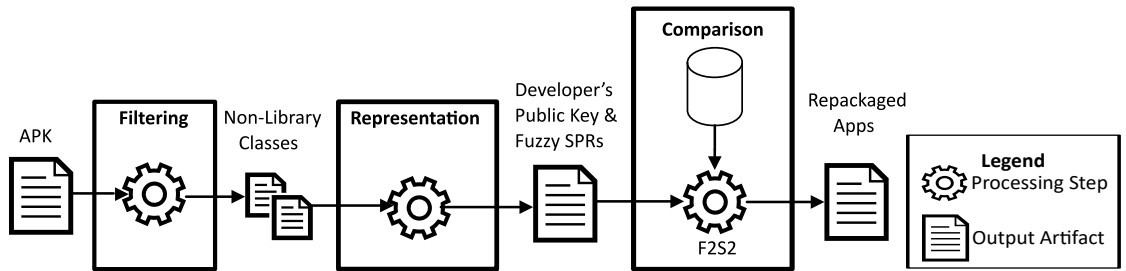


Figure 5.1.: Toolchain to Identify Repackaged Apps

Filtering In the beginning, *CodeMatch* uses a library-filtering tool that reports detected library classes. These classes are then removed from the APK’s codebase. If the library filtering reports multiple library classes in the same app package, *CodeMatch* removes the whole package instead of the individual classes to avoid library classes that cannot be detected with the library filtering.

In addition to the library code removal, *CodeMatch* filters two kinds of apps. First, those apps that consist mainly of library code, plus at most ≈ 300 lines of glue code [rev17]. We consider these apps as non-repackaged because they use only the same libraries as other apps, and the glue code represents only a pattern for the same usage of these libraries. We refer to such apps as *library apps*. Second, we filter apps that are generated using “App-makers”. App-makers generate apps by processing user-created UI-designs. Apps generated by the same App-maker generally share a similar code base without necessarily being repackaged. We filter these apps using a white list of 40 commonly used prefixes of known App-maker frameworks. This list is the result of a web search for Android App-maker frameworks. The list is sufficient to filter generated apps since the Android signing process requires these prefixes, and therefore obfuscators cannot manipulate them.

Representation In contrast to library code that obfuscators may slice when an app only uses a part of the library’s functionality, the app code is likely completely included in a repackaged app because slicing would break its functionality. Since we already removed library code from our target APK, we assume that the remaining code almost entirely corresponds to the potentially repackaged app’s code. As a result, we can use the identified app code to compare it with the code of other apps. To this end, *CodeMatch* represents each app class using its Android-API type list (as defined for our NR in Section 3.2.1), the type list of all its fields, and the SPR for all its methods. We chose SPR instead of Fuzzy SPR because *CodeMatch* calculates the fuzzy hash for all class representations to compare the entire app code with one fuzzy hash.

We address *Code Reordering* by sorting the fields according to the type lists and the methods according to their instruction count. Furthermore, to address *Name Mangling* of package and class names, i.e., to have a name-independent order of classes, we also sort the entire classes by their size, i.e., the sum of sizes of each field, the number of methods, and the instruction count. We address the remaining smaller differences that might have been introduced by obfuscation by fuzzy hashing [Kor06] the entire representation.

Comparison Before we compare potentially repackaged apps, we establish a threshold for the fuzzy-hash similarity, above which we report analyzed apps as repackaged. We determine this threshold by executing *CodeMatch* on 1,000 apps and searching for the best F_1 -measure (harmonic mean between precision and recall). We get the best F_1 -measure with a threshold of 30%.

In order to find a potential repackaged app efficiently, *CodeMatch* builds a database of known apps that it processed as described in the previous steps. Additionally, it indexed the public keys of the respective developers to avoid reporting apps from the same de-

5. Repackage Detection

veloper as repackaged apps. Furthermore, it compares apps from different developers by the fuzzy-hashed representation of the apps' code using *F2S2* [WSY13], which efficiently finds similarity matches based on the edit distance between fuzzy hashes. If the similarity score between the target app and an app in the database exceeds our threshold, *CodeMatch* reports it as a potential repackaged app.

5.4. Evaluation

The evaluation answers the following research questions:

RQ1: How effective is *CodeMatch* compared to other repackage detection approaches?

RQ2: How many apps does *CodeMatch* detect in the wild?

For the experiments, we used a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. We executed the analyses using OpenJDK1.8 212 64-bit VM with 20 GB of heap memory, MySQL 5.7 for the library database, and F2S2 for the app database.

5.4.1. Comparison Against Other Tools

For the evaluation of *CodeMatch*, we collected all app descriptions and the respective developer's public keys (to differentiate between them) from an archive of the Google Play store [pla17b] and fuzzy-hashed the descriptions of each app with *SSDEEP* [Kor06]. Afterward, we compared all fuzzy-hashed descriptions pairwise and randomly selected 1,000 app pairs with different public keys and at least 90%-similar descriptions.

To identify which of these pairs are actual repackaged apps, we installed and executed the app pairs on the emulator *LeapDroid* [lea17]. Subsequently, we manually tagged them (as truly repackaged or not) by checking their similarity in the following process:

1. First, we checked whether the loading screen and main view have the same structure and the same icons.
2. In case of doubts, we checked the actions that a user can perform from the main view.
3. If we were still not sure about the tag, we generated fake accounts, installed needed additional software, and performed all possible actions.
4. If the above steps were insufficient, we also performed a visual inspection of the decompiled code. If the code of both apps was similar, we classified the apps as repackaged; otherwise, they were, in business terms, a "me-too" product.

Following this process, we manually identified 377 app pairs as actual repackaged (true positives) and used all 1,000 app pairs to evaluate the precision and recall of *CodeMatch*,

FSquaDra [ZGC⁺14], *ViewDroid* [ZHZ⁺14], *DroidMOSS* [ZZJN12], and a centroid-based approach [CLZ14] to which we refer as *Centroid*.

To assess the effect of the different library detection approaches on the overall repackaging detection, we removed all *library apps* using *LibDetect*, as described in Section 5.3. These apps are generally falsely identified as repackaged apps by the other approaches as they have no specialized support for this kind of apps, and we want to avoid such biases. We exclude library apps that had mostly less than 300 lines of code, excluding library code, because it is practically impossible to determine whether such apps are illegitimate repackaged apps. For example, most wallpaper apps only differ in the image but are generated apps, not repackaged ones.

After that, we executed all repackaging detectors that use some library detection with all of the following library-detection approaches: *AppSeparator* (AS), *LibDetect* (LD), and the *Common Libraries* white list (WL). We counted it as a false negative if a repackaging detector was unable to classify a repackaged app.

Results Figure 5.2 presents the precision and recall of the different combinations of repackaging and library detection tools. The results are grouped by the repackaging detection approaches and sorted by their F_1 -measure.

All approaches that use white lists perform worst compared to the same repackaging detectors with other library detectors. Using *AppSeparator* improves the F_1 -measure for each approach, but it filters out too many classes of small apps so that it cannot achieve the best results. However, in manual analysis, we discovered that *AppSeparator* is very accurate for large apps.

The results of Figure 5.2 show that the combination of *LibDetect* with any repackaging detection approach produces a precision of 86%, which is better than all combinations of the respective repackaging detection approach and any other library detection.

Observation 8 *Based on the F_1 -measure, CodeMatch + LibDetect is the most effective combination and outperforms all other analyzed tools (RQ1). However, we can conclude that CodeMatch with any library-detection approach performed at least as good as the combinations of the respective library detection with any other repackaging detection approach.*

Overall, *LibDetect* leads to significantly better repackaging-detection results and significantly improves the results of *Centroid* when compared with the initial results. Nevertheless, *CodeMatch* + *LibDetect* gives the best precision and recall.

5.4.2. App Data-Provision & Insights

We used *CodeMatch* to assess the problem of repackaged apps in the wild. For that, we downloaded 46,537 apps from five different Android app stores and analyzed the number of repackaged apps across individual stores. Table 5.2 shows the distribution of the apps across the stores. We obtained the APKs from AppChina [App17b], HiApk [HiA17], and Freewarelovers [Fre17] using *DroidSearch* [RAK⁺15].

5. Repackage Detection

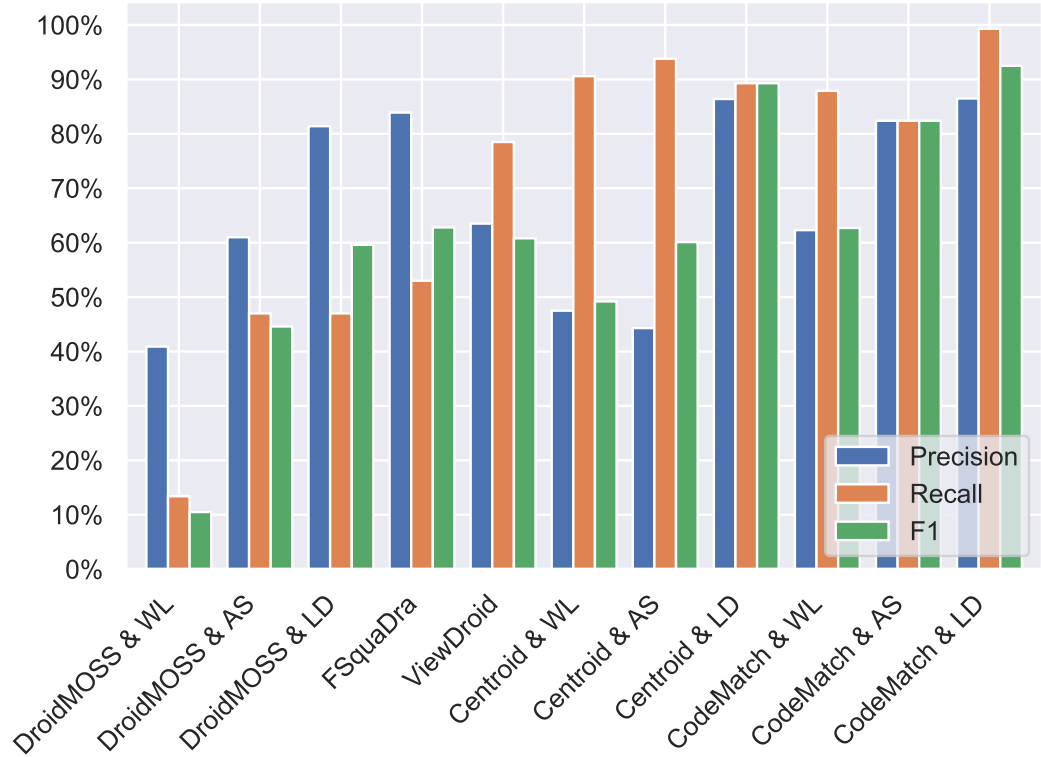


Figure 5.2.: Average Precision, Recall and F_1 -Measure of the Different Repackaging Detection Approaches

Table 5.2 reveals that up to 5.7% of the apps are created using App-makers (see Section 5.3) and—depending on the store—between 2.1% and 68.8% of the apps are *library apps* with less than 300 lines of app (glue) code. We filtered these apps before computing the number of repackaged apps.

Observation 9 Overall, *CodeMatch* identified 7,291 (15.7%) of the 46,537 apps are repackaged (**RQ2**). The problem seems to be most relevant for the Google Play store; it contained in 2017 nearly 20% repackaged apps.

5.5. Threats to Validity

In the following, we discuss threats to validity related to our repackaging detection experiments.

The database of *LibDetect* may not contain all libraries used by apps in our evaluation datasets. In this case, *LibDetect* may fail to identify some library code, and *CodeMatch* may include this library code in the repackaging detection. These missing libraries could lead to other results than the reported ones, but – given the size and quality of the data set – the overall error should be negligible.

Table 5.2.: Analyzed Android Apps from Five App Stores

App Store	Apps	Lib Apps	App Maker	Repackaged
Anzhi [Anz17]	18,889	1,707 9.0%	72 0.4%	2,757 14.6%
Google Play [pla17b]	17,751	371 2.1%	1,018 5.7%	3,510 19.8%
App China [App17b]	4,577	1,260 27.5%	21 0.5%	396 8.7%
HiApk [HiA17]	4,472	1,106 24.7%	6 0.1%	608 13.6%
Freewarelovers [Fre17]	848	583 68.8%	0 0%	20 2.4%
Total	46,537	5,027 10.8%	1,117 2.4%	7,291 15.7%

Our ground-truth dataset for repackaging detection may not be representative of apps in general. To mitigate this threat, we chose our sample from the apps of Google Play store, Anzhi, AppChina, HiApk, and Freewarelovers. First, we filtered the set of all app pairs with similar descriptions, as described in Section 5.4.1. Then we selected a random sample of 1,000 app pairs, representing a confidence level of 99% and a confidence interval of 5%. This sampling strategy may introduce biases because repackaged apps with dissimilar descriptions are left out. However, the intent behind repackaging is to get users to install the repackaged app instead of the original app, making it likely that the repackager uses a similar description. Furthermore, one author classified the app candidates as repackaged or not-repackaged through a manual review (cf. Section 5.4.1). It is possible that our primary criterion, the similarity of the apps’ user interfaces, may lead to some wrong classifications. Due to the high effort of reviewing 1,000 app pairs, it was infeasible to confirm the review results by additional reviewers.

There might be other repackaging detectors that perform better than the ones against which we compared our approach. We included the results of an extensive literature review to identify the tools and, to the best of our knowledge, compared against the state-of-the-art tools.

5.6. Conclusion

We presented a repackaging detection approach that relies (1) on new advanced library detection approaches and (2) the fuzzy hashing of the app code to handle advanced code obfuscations. Additionally, we rely on our different code representations. Each representation abstracts over additional parts of the code to counter more advanced obfuscation techniques.

The evaluation demonstrated the effectiveness of *CodeMatch* by applying it to real-world apps taken from Android app stores. We were able to determine that 15% of the apps are in repackaged form across different app stores.

Summary

In Chapter 3, we presented *LibDetect* our approach for detecting libraries in obfuscated apps. It uses five representations for its detection algorithm; each abstracts more information from a method code to identify more obfuscated methods without losing precision. After the matching of methods, *LibDetect* aggregates the methods to entire library classes.

For the evaluation of *LibDetect*, we compared its precision, recall, and F_2 -measure with an approach that uses a white list of *Common Libraries* [LKLT⁺16] and a more advanced library-detection approach, called *LibRadar* [MWGC16]. To compare these approaches, we downloaded 1,000 apps from different app stores and manually identified all library classes. The comparison showed that *LibDetect* identifies more than six times more library classes than the other approaches.

Since many approaches need to separate app code from library code without the detection of libraries, we introduced in Chapter 4 *AppSeparator* that separates app code without the need for a database of all library-method representations. The design of *AppSeparator* is based on the single compilation assumption, which states that libraries were already compiled before their integration into apps; therefore, libraries have no dependency on any specific app. Using the intuition of this assumption, *AppSeparator* performs the following steps to separate app code.

First, it extracts all classes containing entry points and all classes from the main package of an app. With these classes, *AppSeparator* transitively identifies all additional classes that access the ones identified earlier.

Second, since obfuscation can blur the lines between app code and library code, *AppSeparator* calculates for each class the FanIn and FanOut metrics.

Last, *AppSeparator* uses a classifier that is trained using the fan-in and fan-out sets of the classes to decide whether a class belongs to the app code.

For the evaluation of *AppSeparator*, we used the 1,000 apps from the evaluation of *LibDetect* and split them in 80% training and test set and 20% validation set. We used the former set to train our classifier and the latter to compare *AppSeparator* with *LibDetect* and *LibRadar*. The results suggested that *AppSeparator* is at least 11.55% more accurate in separating app code than the other approaches. In an additional experiment, we discovered that the usage of the classifier is, to a significant degree, responsible for the better results of *AppSeparator*.

Chapter 5 introduces our repackaging detector *CodeMatch* and compares the ability of *AppSeparator* and *LibDetect* to filter out library code. *CodeMatch* first filters out small apps and apps generated by App Makers [app17a] and compares the remaining apps with a database of original apps. If one app has a high similarity to one in the database and is not produced by the same developer, it is repackaged.

5. Repackage Detection

Before *CodeMatch* compares an app with the database, it also filters out the library code using either *AppSeparator* and *LibDetect*. Because library code influences the similarity measurement of apps, it is essential to remove as much code as possible. After the filtering, *CodeMatch* uses the most abstract representation from *LibDetect* to evade obfuscation techniques used by the repackagers.

For the evaluation of *CodeMatch*, we analyzed manually 1,000 app pairs based on their appearance and their code. Afterward, we compared *CodeMatch* with *FSquaDra* [ZGC⁺14], *ViewDroid* [ZHZ⁺14], *DroidMOSS* [ZZJN12], and *Centroid* [CLZ14]. The results showed that *CodeMatch* using the library detection *LibDetect* outperformed all other approaches at least by 3.58%.

Finally, we discovered, in an additional experiment, that at least 15% of the apps in different markets are repackaged.

Part II.

Obfuscated Names

Obfuscated Name Detection and Recovery

Developers apply name obfuscation to apps to protect their intellectual property. However, it is also used by attackers to conceal repackaged apps and malicious payloads in these apps. Previous work proposed several approaches [BRTV16, RVK15, VCD17, Jaf17, AZLY19, CFPK20] to support analysts in understanding code entities with obfuscated names, such as class, field, method, and variable names. These approaches automatically learn names from large code samples and suggest learned names to replace obfuscated ones. For the replacement of names, they analyze a code entity’s surrounding context to suggest a name that was learned together with a similar context. This context is captured using dependency networks, Abstract Syntax Trees, or by treating the code as text blocks. The approaches extract the context and train a model to associate structures with the code entities’ names. Because these approaches infer names from code structures, we refer to them in the following as name inference approaches.

While these approaches foster analysts’ understanding, they have two main drawbacks: First, most name inference approaches do not analyze if the code samples used for their learning process contain obfuscated names. The names they suggest depend heavily on the code samples they use during their learning process. If the code entities in these samples have obfuscated names, they would suggest these obfuscated names during their inference process. Since most approaches learn from code in open-source apps, it could be assumed that these approaches are not affected by obfuscated names. However, even code from open-source apps may integrate libraries containing obfuscated names. Some libraries are only released in obfuscated form; therefore, the names of these library code entities are not suitable for name inference approaches because the names are not available in the non-obfuscated form. Additionally, during the inference process of an app, the approaches do not identify which code entities of that app have obfuscated names, leading to the replacement of non-obfuscated names. As a result, some approaches that should support analysts to deobfuscate names suggest obfuscated names for replacing non-obfuscated ones.

Second, the approaches do not distinguish between library code and app code and may suggest the names of library code entities to app code entities. This situation can occur if a code entity’s context in the main code of the app is similar to the context of a code entity from a library. Names of app code fundamentally differ from library code entities because the names of app code entities are tailored for a specific use case. In contrast, the names of library code are more generic. Since every app consists of at least 60% library code [WGMC15], the names of library code entities occur more frequently than the names of app code entities. Because name inference approaches learn only names that occur in the majority of the code samples, they learn mostly names of library code entities and may suggest these names for app code entities. For instance, some developers

may not obfuscate their app codebase, but their apps contain obfuscated libraries. A name inference approach would identify suitable names for the library code entities but also suggest library code entities' names to app code entities.

To support analysts, we present two approaches. The first approach pinpoints obfuscated names in the entire codebase of an app, and the second one focuses on the recovering of names for library code entities. For the recovering of obfuscated names for library code entities, we introduce in Chapter 6 *LibMapper* that can map original names of library code entities to their obfuscated counterparts. For the remapping, it uses *AppSeparator* (Chapter 4) to separate library code from app code, which leads to a more reliable suggestion of names only for the library code of an app. Afterward, *LibMapper* uses *LibDetect* (Chapter 3) to get possible name suggestions for the mapping of classes. With *LibMapper*, malware analysts can uncover which library code entities are used by payloads and investigate whether a payload exploits unknown vulnerabilities of these code entities. In the following chapters, we refer to the process of mapping original names of library code entities to their obfuscated counterparts as library mapping.

Chapter 7 introduces *ObfusSpot* which identifies obfuscated names in three steps. In the first step, it identifies obfuscated names whose naming schemes differ from non-obfuscated names. In the second step, it analyzes name frequencies to identify obfuscated names that occur very frequently. In the last step, *ObfusSpot* uses *LibMapper* to identify patterns in obfuscated names of library code entities and use these patterns to uncover obfuscated names in the rest of the code. In contrast to previous approaches [Mir18, WR17, PYC⁺19, KNGS18, Don18, WHA⁺18], *ObfusSpot* can pinpoint obfuscated names of code entities instead of just pointing out that an app contains obfuscated names.

In the end, we summarize all contributions of this part.

6. Library Mapping

As library mapping, we describe the process to map the original names of library code entities, such as class, field, and method names to their obfuscated counterparts. The names of these code entities are essential to understand their functionality during the analysis of an app. However, these names are manipulated by developers to protect their intellectual property and by malware authors to hide their payload.

In order to support security analysts and avoid the mapping of code entities by hand, previous works [YDGPS16, Jav20b, Jav17, RVK15, BRTV16, VCD17, ABBS14, ABBS15] introduced several approaches to learn names of code entities and map them to their obfuscated counterparts that have a similar surrounding context as the code entities from the learned names. This context is captured using dependency networks, Abstract Syntax Trees, or by treating the code as text blocks. The approaches extract the context and train a model to associate structures with the code entities' names. Despite the usefulness of these approaches, they have various drawbacks. For instance, some approaches operate solely on source code, which is rarely available for apps. Additionally, these approaches do not distinguish between library and app code and assign library code names to app code entities and vice versa. Since most of the code in an app belongs to libraries [WGMC15], the approaches learn more names of library code entities, and it is more common that these names are assigned to app-code entities [GS10].

In this chapter, we introduce *LibMapper* that operates on Java bytecode, which can easily be extracted from Android apps. It uses *LibDetect* and *AppSeparator* as preliminary analyses and assigns to each library code entity the name of its original counterpart. While *LibMapper* uses *AppSeparator* to identify which classes belong to library code, it needs *LibDetect* to get suggestions of possible name candidates for matching library classes. As explained in Section 3, *LibDetect* identifies potential candidates for library methods and uses the best mapping of these methods to identify their potential declaring classes and the library version. While we already used the potential declaring classes to filter out library code in Section 5, we can use all potential library methods to get a set of potential declaring classes candidates that can be used for library mapping. Given a library class in an app under analysis, *LibDetect* identifies all methods and derives from them a set of potential library class names. This set of class names is used by *LibMapper* to determine the name of each field and method member. For instance, if *LibDetect* returns for a class in an app, ten potential class names, then *LibMapper*'s database contains all names of the associated fields and methods of each class name. These field and method names are used to map as many obfuscated code entities as possible. Afterward, given the context of the code entities, *LibMapper* determines from all potential name mappings, the one that maps most of the names to assign this mapping to the library class and its members.

6. Library Mapping

With this technique, *LibMapper* does not suffer from the drawbacks of previous approaches, and by using *LibDetect*, it can resist advanced name obfuscation without overwriting non-obfuscated names. Standard configurations of obfuscators replace meaningful names with a short sequence of random characters. However, more advanced name obfuscations may repackage classes into a single package or use a natural language dictionary for their replacement procedure to make the identification of obfuscated names more challenging [con19a].

To avoid the replacement of non-obfuscated names, *LibMapper* checks whether the class under analysis has the same name as one of the candidate names produced by *LibDetect* and integrates them with a higher probability into its mapping procedure.

While the method representations of *LibDetect* are designed to determine the exact version of a library, they may not be suitable to provide a method name that remains equal across multiple versions of libraries. For this reason, we analyze the stability of the method representations across class boundaries by measuring how often the same method representation is used in different classes without belonging to the same library. We refer to this measure in further descriptions as the collision rate and use for *LibMapper* a set of method representations with another collision rate than for *LibDetect*.

For the evaluation, we compare *LibMapper* with the state-of-the-art deobfuscator *DeGuard* [BRTV16] that assigns each code entity, including app and library code, a new name. To show both approaches' effectiveness, we measure their performance on apps manipulated with *ProGuard*'s standard configuration for name obfuscation and the most advanced one [Pro17].

In the following sections, we present our related work in Section 6.1, a case study whether current representations are suitable to map names (collision rate) in Section 6.2, followed by our approach *LibMapper* in Section 6.3. Afterward, we evaluate *LibMapper* in Section 6.4, discuss threats to the validity of our results in Section 6.5, and conclude our work in Section 6.6.

6.1. State-of-the-Art of Library Mapping Approaches

In this section, we describe the state-of-the-art approaches for library mapping. All approaches are either rule-based or use artificial intelligence to infer names from previously learned contexts.

Rule-Based Approaches Early attempts inferred names based on associations of structures and were primarily rule-based. Approaches such as *DREAM++* [YDGPS16], *JavaDecompiler* [Jav20b], and *JavaDeObfuscator* [Jav17] used known structures like loop variables, constants, classes, fields, or methods to assign frequently used names such as `i` for obfuscated loop variables or type-based names such as `str` for field names with the type `java.lang.String`. However, an analyst could not use these approaches to map libraries because most libraries had specific names for their code entities, and rule-based approaches use only very generic names.

JSNice [RVK15] uses conditional random fields [LMP01] (CRF) to infer identifiers based on JavaScript’s pre-defined language constructs. First, it extracts all associations between local variables and learns which names are used along with these variables. The inference phase also extracts the same associations and matches these associations with previously learned ones to assign the learned variable names to the obfuscated ones. Instead of focusing on one variable at a time, the approach uses an approximate Maximum a Posteriori algorithm [KF09] (MAP) to adjust all names to a global maximum.

JSNice operates only on local variables and is not suited for class, method, or field names. In addition to the limited usability, the approach assigns names of local variables based on language constructs without considering non-obfuscated names. That way, the inference overwrites previously known and descriptive names. While *JSNice* could be used to infer names in JavaScript functions, it is not suited to identify libraries and map their names in Android apps.

DeGuard [BRTV16] operates on the same platform as *JSNice* and uses CRF for library mapping. In contrast to *JSNice*, *DeGuard* does not establish associations between local variables. It establishes them between package names, class names, method names, field names, and constants by using the API of the Android class library instead of language constructs as a basis for the associations. As a result, *DeGuard* can infer names for the above-mentioned code entities by using the approximate MAP estimation of associations from previously learned code entities.

Since *DeGuard* does not identify obfuscated names, it suffers from the same drawback as *JSNice* that overwrites previously known and descriptive names. In addition to this, the algorithm identifies libraries by using only a white-list of library package names and hence is vulnerable to *Name Mangling*.

JSNaughty [VCD17] builds a statistical machine translation [Koe09] (SMT) model of non-obfuscated JavaScript code and infers identifier names of obfuscated code by using the identifier’s surrounding language constructs. *JSNaughty* blends the capabilities of *Autonym* [VCD17], a tool designed by the same authors with *JSNice* [RVK15]. *Autonym* learns a language and translation model for SMT.

JSNaughty builds names from language constructs to all local variables without considering non-obfuscated names so that its inference procedure overwrites previously known and descriptive names. Furthermore, it overwrites app code-entities with names from the library code. Finally, *JSNaughty* does not identify libraries and suffers from the same drawbacks as *JSNice*.

NATURALIZE [ABBS14] identifies the styling-conventions used by Java developers and suggests the learned conventions if another developer adds new code structures to keep a uniform style in code reviews.

NATURALIZE is not suited for obfuscated apps, because obfuscators manipulate all tokens that are used by it. Additionally, *NATURALIZE* is neither able to handle bytecode nor to identify libraries for library mapping.

6. Library Mapping

Allamanis et al. propose an approach [ABBS15] that builds upon *NATURALIZE* by dividing identifiers into subtokens, and inferring for each subtoken a more suitable name. For this purpose, the approach compares the surrounding context with a previously learned context model. The approach builds the context model by extracting the subtokens of surrounding Abstract Syntax Tree (AST) nodes and other identifier-related features in the Java source code.

Nevertheless, this approach’s features depend on the AST and source code that are no longer available after the compilation and obfuscation of an app. Additionally, this approach overwrites previously known and descriptive identifiers by names from its learned model. Finally, the approach does not identify libraries and is therefore not suitable for library mapping.

Code2Vec [AZLY19] extracts paths from an AST of a method. Using the extracted paths, it learns the most relevant features of a path for a given method name. Finally, it uses the learned features to infer method names in similar environments.

Since *Code2Vec* operates only on source code and needs a complete AST, it is not suitable to map obfuscated code. This conclusion derives from the fact that *Code2Vec* uses the names of other code entities to infer method names. Thus, if obfuscators manipulate the AST and all names in a method, *Code2Vec* cannot infer correct names [CFPK20]. Additionally, it overwrites names that have not been changed by the obfuscator.

Discussion Summarizing the discussion above, we need a better library mapping that can correctly identify library instances without overwriting non-obfuscated names. Currently, only *DeGuard* can identify library instances of Android bytecode. Many approaches operate only on source code, and others use variable names without considering field, method, and class names. Although *DeGuard* can identify library instances, its library detection is based on a white-list and, therefore, is vulnerable to changes in the package hierarchy. These limitations lead to a poor recall, as our empirical comparison of *DeGuard* against our new approach will reveal in Section 6.4.

6.2. Case Study of Collision Rate from Method Representations

In this section, we discuss the suitability of method representations for library mapping. For this purpose, we analyze the representations used by several library detection techniques that use these representations to determine library classes or a library version. Library detection approaches match libraries either by using method representations [WWZR18, BBD16, ZDZ⁺18, FR19] and combining these representation to entire classes (like our work in Section 3.2) or by directly using a representation for classes or packages [LWW⁺17, ZBK19, MWGC16].

While *Code Slicing*, *Code Restructuring*, and *Code Optimization* may evade representation for classes and packages, method representations are used as a fallback to match library parts if the other representations failed to match a class or package.

6.2. Case Study of Collision Rate from Method Representations

Previous works measure the robustness of their method representations [WWZR18] or the uniqueness of the entire profile [BBD16, ZDZ⁺18] without considering that one method representation can occur in different locations and lead to wrong assignments of names to code entities. We measure how often these representations collide with ones from different locations and further refer to this measurement as the collision rate. To analyze the collision rate of different representations, we re-implemented the method representations of *LibScout* [BBD16] and *Orlis* [WWZR18] and compared them with our five representations from Chapter 3.

The collision rate was calculated by measuring one minus the occurrence ratio of a representation to the number of its method declaration in different classes. The following formula shows the collision rate:

$$\text{collision rate} = 1 - \frac{1}{\# \text{ of classes declaring such representations}}$$

For instance, if a tool uses the same representation for three methods that are declared in three different classes, the collision rate is $1 - \frac{1}{3} = 66.66\%$.

To enable an accurate calculation, we need to ensure that each method representation in a data set is unambiguously connected to its class. This connection cannot be ensured by straightforwardly using code from different apps because these often use the same library code but are changed by obfuscation. As a result, we acquired the data set described in the next section, which was not transformed by any obfuscation technique.

6.2.1. Data Set

To acquire a data set of apps that do not contain obfuscation, we collected all 1,879 apps from the F-Droid store [F-D19] and analyzed their code. The Android developer website [pro20a] recommends to integrate *ProGuard* [Pro17] in the developers' build process so that the published apps contain obfuscated names. However, since developers often use the standard configuration of *ProGuard*, which transforms meaningful names into a short sequence of random characters, we can easily identify them by exploiting the shortness of the names.

For the filtering, we compared the average class-, method-, and field-name length per package with the average length of code entities in non-obfuscated packages. If an app had a package that contains many short names, we excluded this app. After this step, only 453 apps remained, and we checked them for the usage of obfuscation manually. In total, we acquired 302 unique, non-obfuscated libraries that we used to compare the collision rate for the seven representations.

6.2.2. Results

Given the 302 libraries, we extracted our five representations from Section 3.2.1, *Orlis*' representation with transitive method calls, and *LibScout*'s fuzzy method signatures. Afterward, we stored in a database the representation of each method connected with

6. Library Mapping

the class that declares the corresponding method. Using this database, we calculated the collision rate for each representation.

Figure 6.1 shows the collision rate of all seven method representations categorized by the number of instructions. The x-axis shows the number of instructions in the logarithmic scale to the base two. We included the method representations whose instruction number ranges between two x-values to the larger value. The figure lists our five representations from Section 3.2.1 (BC, AR, NR, SPR, Fuzzy SPR), the representation of *Orlis* (CallR), and *LibScout*'s fuzzy method signature (Fuzzy Signature). Additionally, we added the percentage of methods that have a particular method length (Distribution).

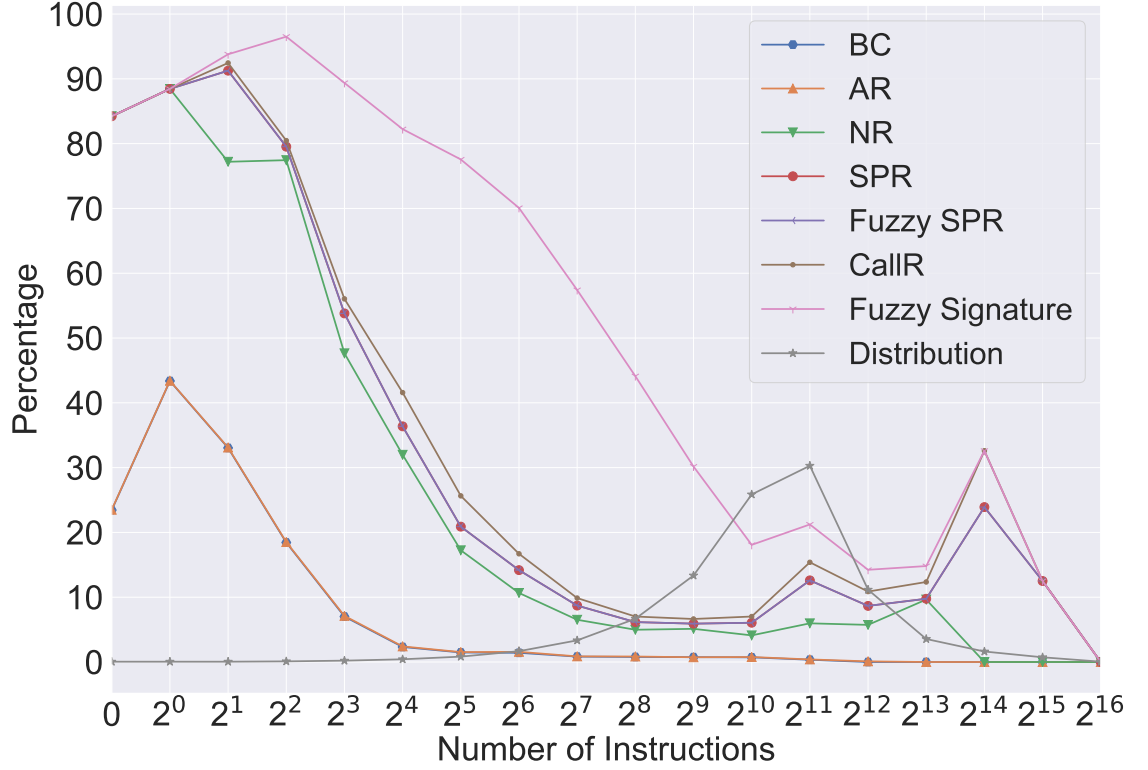


Figure 6.1.: Collision rate categorized by number of instructions

Given all collision rates, we have made the following observations:

In Figure 6.1 all collision rates decrease with the increase of the instruction count and increase with a higher method distribution. Interestingly, five of the representations have the same collision rate, with either zero or just one instruction; this results from the fact that these methods have too few instructions to differentiate them from plain fuzzy signatures as used by *LibScout* [BBD16] and *LibPecker* [ZDZ⁺18]. Only BC and AR have a lower collision rate because they use the original method signature in their representations.

These fuzzy signatures have the highest collision rate across all categories of the instruc-

tion counts. Its peak is at 3 to 4 instructions (2^2 instructions), and such methods are often either **getters** or other methods that directly delegate all parameters to another method.

The collision rates of *CallR*, the *SPR*, and, without considering partial matches, of *Fuzzy SPR* are very close to each other. While the *Fuzzy SPR* and the *SPR* overlap entirely, the *CallR* has a slightly higher collision rate because it does not use all instructions in its representation but only those that call methods transitively.

The *NR* has a lower collision rate than the previously described representations because it only removes all names without changing any instruction. As shown in Figure 3.4 of Chapter 3, the removal of names is already sufficient to match more than 95% of the obfuscated methods.

Most surprising are collision rates of BC from methods smaller than eight instructions. BC includes all unchanged instructions and the unchanged method signature. These high collision rates are caused by default constructors, **getters**, interface, and other methods that are not distinguishable among different classes. Obfuscators such as *ProGuard* do not change the signature of default constructors. However, as shown by our experiment, these methods alone are not suitable for identifying obfuscated classes.

Figure 6.1 also shows that the collision rate of the *AR* overlaps with BC. This overlap is not surprising since only the modifiers and program counters are removed from the bytecode to create the *AR*. While the *AR* has a low collision rate, it can only be used to match non-obfuscated methods that differ due to access restrictions of methods used in different library versions.

Using the percentage of the method distribution, we identified that with a higher distribution, the collision rate drops rapidly below 10% starting at 2^6 . However, this effect may also arise from the complexity of the method representations that incorporate more information with a higher number of instructions.

From Figure 6.1, we can deduce that matching all representations in ascending order according to the collision rates yields the best results. However, due to the various overlaps between different representations, we use only **AR** instead of **BC** for non-obfuscated methods, and **CallR** instead of **SPR** because it is more robust against obfuscation.

With the gained knowledge, we design our library mapping approach *LibMapper* in the next section. Which organizes the representations in the following order: **AR**, **NR**, **CallR**, **Fuzzy SPR**, and **Fuzzy Signature**.

6.3. LibMapper

In this section, we present our approach *LibMapper* that identifies not only library classes or packages but also maps each original class, method, and field name to the obfuscated

6. Library Mapping

counterpart. *LibMapper* follows the steps shown in Figure 6.2 and uses for *LibDetect* the method representations described in the previous section to identify a set of potential class name candidates.

First, *LibMapper* uses *AppSeparator* (cf. Chapter 4.2) to separate the library code from app code. Second, it matches a set of potential classes to the library code using either a representation for classes or methods. Third, our approach calculates the likelihood of the class, field, and method data fitting the potential class-matches. Fourth, *LibMapper* combines all probabilities to resolve the mapping for each code entity. It also considers preferred classes that result from previously matched ones. Finally, *LibMapper* returns a mapping file that contains all code-entity mappings that an analyst can use to rename all names in an app using *ProGuard* [Pro17].

In the following, we describe the usage of our representations and the interaction between different processing steps.

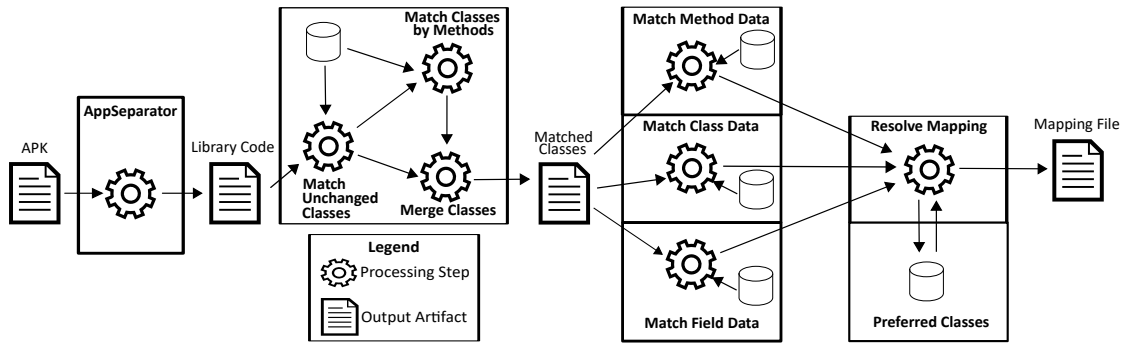


Figure 6.2.: Process to resolve class, method, and field library mappings

6.3.1. AppSeparator

We use *AppSeparator* to identify which classes belong to library code in preparation for the next processing step. *AppSeparator* has shown to be adequate to separate app code from library code in Section 4.3.

6.3.2. Match (Structurally) Unchanged Classes

Given the potential library classes identified by *AppSeparator*, we match library classes that have not been structurally changed to identify them more efficiently. For this matching, we use two representations that we previously stored for each library class. The first representation consists of the following class, field, and method information, which we concatenate to a fingerprint of the class.

Classes: For classes, we remove all modifiers and concatenate the name of the class, followed by the superclass's name, and the names of interfaces sorted by their names to the result string.

Fields: For fields, we remove all modifiers of the fields, sort them by their names and append each field with its type name and field name to the class string.

Methods: For methods, we extract the *AR* for each method, sort the methods by the representation, and append them to the class string.

To enable a fast comparison, we hash the resulting string with SHA1 and refer to the hash sum in the following as *Class AR*. As we have shown in Section 6.2, the *AR* has the same collision rate as the bytecode; however, since the *Class AR* is more robust against changes to modifiers, we use this representation to match structurally-unchanged classes.

Since the *Class AR* is vulnerable to *Name Mangling* and *Fake Types*, we manipulate the classes, fields, and methods and concatenate their strings for the second representation in the following way:

Classes: For classes, we use the first type names from the inheritance hierarchy, which belong to the Android API as a class string similar to parameters and return types in Section 3.2.1 (*NR*). For instance, if class *X* extends class *Y* and class *Y* extends class *Z* that belongs to the Android class library, then we use the class name of class *Z* in our class string.

Fields: For fields, we remove all modifiers and field names. Afterward, we transitively iterate over the type hierarchy of a field type and replace the field type name with the first type name that belongs to the Android API. Finally, we sort the fields by these names and append each field with its type representation to the class string.

Methods: For methods, we extract the *NR* for each method, sort the methods by the representation, and append them to the class string.

As done with *Class AR*, we hash the resulting string with SHA1 and use it in the following as *Class NR*. We first match a class with the *Class AR*. If we cannot match a class with this representation, we match it with the *Class NR*. Afterward, we match all remaining, unmatched classes from the library code using the method representations as in the following description.

6.3.3. Match Classes by Methods

We use five method representations in increasing abstraction order to identify a set of potential class matches for classes that were not matched using the class representations. However, we use a slightly different set of representations than *LibDetect* because it has a different focus than *LibMapper*. While *LibDetect* tries to match a method signature as precise as possible to identify the library's version, *LibMapper* tries to match as many names as possible to map each class, field, and method name of a library.

Based on the collision rates of *BC* and *AR*, we do not use *BC* as our representations because *AR* has the same collision rate but is more robust to modifier changes. The same applies to *Fuzzy SPR* over *SPR* so that we use the former but not the latter. With these reductions, only the representations *AR*, *NR*, and *Fuzzy SPR* would remain to match

6. Library Mapping

methods. However, if we cannot match specific methods with these representations, we also use *CallR*, and *Fuzzy Signature* from Section 6.2.

Given this set of representations, we match a set of potential library classes to a library class from an app using the same method-to-class matching, as described in Section 3.2. To this end, we match each of the five method representations extracted from the library code with the same representations stored in the database in the order from most to least precise representation. Using the stored fully-qualified name of the classes and each matched method representation, we aggregate the set of potentially used library classes.

In order to reduce the number of potentially matched library classes, we use heuristics gained from the insights of Section 6.2: We first match all methods with more than 16 (2^4) instructions to avoid mismatches due to huge collision rates. Afterward, we match the remaining methods based on the identified class names of the previously matched methods or use all methods if there are no methods with more than 16 instructions.

6.3.4. Merge Classes

From matching of unchanged classes and classes by method representations, we get a set of potentially fully-qualified names of classes for each library class in an app. While the sets of unchanged classes contain only one or a few matched classes, the ones from method-representation matching may contain several hundred potential library classes. To identify a precise mapping for each class, field, and method name, we combine all possible information of the entities under analysis.

6.3.5. Match Method Data

To map all methods to a potentially matched class, *LibMapper* uses multiple features to calculate for each matched method (m) a score of $m \in [0, 1]$. The features were selected to facilitate the identification of the method name by encompassing its core functionalities and its surrounding code entities without relating on their names. After calculating the scores, *LibMapper* summarizes them and divides the result by the number of features. Finally, *LibMapper* maps each class's method by using the library information with the highest resulting value, which is stored in our database. In the following, we describe the different features that *LibMapper* uses to determine a mapping for a library method.

Method Name: While an obfuscator could manipulate a method name, *LibMapper* uses it to avoid mismatches caused by non-obfuscated names. However, *LibMapper* does not calculate a score for partially matched method names, it uses 1.0 for a complete match and 0.0 otherwise.

Subclass Methods: If an obfuscator did not change a method name, *LibMapper* also checks all methods that override this method in subclasses to reduce time wasted for additional checks and matches the name with 1.0 for a complete match and 0.0 otherwise.

Sibling Class Methods: As for subclass methods, if an obfuscator did not change a method name, then *LibMapper* checks also the name of a method whose declaring class inherits from the same superclass. If the method name matches completely *LibMapper* uses the score 1.0 and 0.0 otherwise.

Number of Instructions: The number of instructions may change by utilizing code manipulation, method inlining, or extraction. However, since most of the method instructions stay unchanged, *LibMapper* uses this number to discriminate between different sizes of methods. To use the method size as score, *LibMapper* uses the number of instructions from the method under analysis ($instMua$) and compare it with the number of instructions of the potentially matched method ($instPmm$) by using the formula $\frac{\min(instMua, instPmm)}{\max(instMua, instPmm, 1)}$ where $instMua, instPmm \in \mathbb{N}$.

Method Signature: Since obfuscators change the names of methods and types that do not belong to the Android class library, we could not use the unchanged method signature to map method names. Nevertheless, if a method signature uses primitive types or types from the Android class library, we could compare the method under analysis and the potentially matched method. Since the partial match of method signatures would lead to an enormous increase of false positives, *LibMapper* compares the method signatures by the exact number of parameters and matches of the following kinds of parameters. If a parameter type belongs to the set of primitive types or a class from the Android class library, *LibMapper* checks whether the type resides at the same index position of the parameter list and is the same type. Otherwise, it checks whether the index positions of both types are equal. However, if both parameters are of type array, *LibMapper* compares the element types of the parameters by using the comparison for non-array types. The same applies to the return type of the method signature. *LibMapper* uses 1.0 for a complete match of a method signature and 0.0 otherwise.

Fully-Qualified Name of the Class: Obfuscators often minify the fully-qualified name of a class. However, they cannot change each class name since other sources reference these names. To avoid matching methods from other classes if the class name is unchanged, *LibMapper* compares the fully-qualified class name of the method under analysis with the potentially matched method. It uses 1.0 for a complete match and 0.0 otherwise.

Fuzzy SPR: Since we have no comparison value whether structures in the method under analysis match the ones from the potentially matched method, *LibMapper* uses the *Fuzzy SPR* to compare the structures. Recall that we constructed the *Fuzzy SPR* using *SSDEEP*, whose comparison function calculates a score between 0 and 100 for each fuzzy hash. Consequently, we use the tool's integrated functionality to get a comparison value between 0.0 and 1.0 using $\frac{SSDEEP\ output}{100}$.

Fuzzy Signature: If obfuscators change the methods' structures so that the method bodies have no similarities, we can still determine the similarity of a method by

6. Library Mapping

comparing their *Fuzzy Signatures*. In contrast to the method signature matching above, *Fuzzy Signatures* also deal with the inheritance and re-ordering of parameter types. Because of this fact, *LibMapper* uses the *Fuzzy Signatures* to determine complete matches (1.0) or none 0.0 otherwise.

Similarity of Calling Methods: Since obfuscators may change the entire code of a method, we use the calls of the method to deduce its name. *LibMapper* compares the methods that call the method under analysis with the method calls from a potentially matched method. However, we do not directly compare the method calls but their signatures. To compare two method signatures of the method under analysis (mua) and the potentially matched method (pmm), *LibMapper* uses their method names, return types, and all parameter types. It calculates for each pair of method signatures the similarity formula:

$$similarity_{ij} = \frac{similar(m_i, m_j)}{\max(\#parametersOf(m_i), \#parametersOf(m_j)) + 2}$$

where $m_i \in \text{All Method Calls of mua}$ and $m_j \in \text{All Method Calls of pmm}$.

The addition with two in the denominator is used for the matches of the method name and the return type. Given the comparison of individual method signatures, we combine them by using the formula:

$$Similarity\ of\ Calling\ Methods = \frac{\sum_i \max_j(similarity_{ij})}{\max(\#pmm, 1)}$$

where $i \in \{0..\#pmm\}$ and $j \in \{0..\#mua\}$.

We calculate this score using only the number of pmm in the denominator because library methods that were integrated into apps might have a different set of callers than those extracted in a separate module.

Similarity of Method Calls: To match the method calls of the method under analysis with the ones of a potentially matched method, we compare the method signatures of the calls analogously to the previous feature, only using the method calls instead of calling methods.

Similarity of Used NR Field-Signatures: While the above features only consider method code and calls, this feature uses fields that are read and written by the method under analysis. As obfuscators may change the name of fields, we use the field types and transform them, analogously to method parameter types, by using the *Nameless Representation* (NR). The NR replaces a type that does not belong to the Android class library with a set of its transitively inherited Android class library data types (cf. Section 3.2.1). We calculate the similarity of NR field-signatures by using the following formula that is similar to the one from *Similarity of Calling Methods*. It uses only the NR field-signatures (nrf) of the method under analysis (nrfMua) and the potentially matched method (nrfPmm):

$$field\ similarity_{ij} = similar(nrf_i, nrf_j)$$

where $nrf_i \in \text{All Fields of } nrfMua$ and $nrf_j \in \text{All Fields of } nrfPmm$.

$$\text{Similarity of Used NR Field Signatures} = \frac{\sum_i \max_j(\text{field similarity}_{ij})}{\max(nrfPmm, 1)}$$

where $i \in \{0..\#nrfPmm\}$ and $j \in \{0..\#nrfMua\}$.

Similarity of Methods Connected By Fields: Although matching NR field-signatures introduces a feature for fields into the comparison of methods, it may provide too little information because the field types may not differ between different methods. For instance, a data container class could contain many fields with the type `int`, which are accessed by many getter and setter methods so that these fields cannot be differentiated using these methods. To tackle this issue, we use instead of field signatures, all method signatures (ms) that access the fields which are also accessed by the method under analysis ($msMua$) and the potentially matched method ($msPmm$). For the comparison, we use the following formulas:

$$\begin{aligned} \text{similarity of field - connected methods}_{ij} = \\ \frac{\text{similar}(ms_i, ms_j)}{\max(\#parametersOf(ms_i), \#parametersOf(ms_j)) + 2} \end{aligned}$$

where $ms_i \in \text{All Methods Connected By Fields from } msMua$ and $ms_j \in \text{All Methods Connected By Fields from } msPmm$.

$$\begin{aligned} \text{Similarity of Methods Connected by Fields} = \\ \frac{\sum_i \max_j(\text{similarity of field - connected methods}_{ij})}{\max(msPmm, 1)} \end{aligned}$$

where $i \in \{0..\#msPmm\}$ and $j \in \{0..\#msMua\}$.

6.3.6. Match Field Data

After matching the method data, we extract the following features to match fields for a potentially matched class. Since the representation of fields provides limited information to identify their names, we use to encompass the fields not only their representation as features but mainly information from methods that access the fields. As with the method data, we assign a field mapping for each potentially matched class using the maximum of the combined scores from the following items.

Field Signature: Since obfuscators may not be able to change all field names, *LibMapper* compares fields by using their signatures and uses 1.0 for a complete match of a field signature and 0.0 otherwise.

Fuzzy Field Signature: For all fields whose names were changed by obfuscators, *LibMapper* compares the field types using the *Nameless Representation* and uses 1.0 for a complete match of a field NR and 0.0 otherwise.

6. Library Mapping

Similarity of Method Accesses: Since the *Fuzzy Field Signature* may not suffice to match fields with common types, *LibMapper* uses the methods that read or write each field to identify it. *LibMapper* matches each method that accesses the field under analysis (mFua) with all methods that access the potentially matched field (mPmf) and divides it by the number of accessing methods for the fields (namFua, namPmf) using the formulas:

$$\text{Method Access Similarity}_{ij} = 1 - \frac{|mfua_i - mPmf_j|}{\max(1, mFua_i, mPmf_j)}$$

$$\text{Similarity of Method Accesses} = \frac{\sum_i \max_j(\text{Method Access Similarity}_{ij})}{\max(\text{namPmf}, 1)}$$

where $i \in \{0..\text{namPmf}\}$ and $j \in \{0..\text{namFua}\}$.

Similarity of Fields Read Before Another Field: Using the methods' field-accesses may be sufficient for most fields. However, if multiple fields are accessed by the same methods, they cannot be differentiated using the previous data. For this purpose, *LibMapper* checks each method that reads the field under analysis, whether another field was read before it using the program counter. *LibMapper* compares each previously read field using its *Fuzzy Field Signature* (fs_i, fs_j) for the number of all reading methods of the field under analysis (nrmFua) and the potentially matched field (nrmPmf). We match the fields either by using 1.0 for a full match or 0.0 otherwise. For the comparison, we use the following formulae:

$$\text{Fuzzy Field Signature Similarity}_{ij} = \text{equal}(fs_i, fs_j)$$

$$\text{Similarity of Fields Read Before Another Field} = \frac{\sum_i \sum_j (\text{Fuzzy Field Signature Similarity}_{ij})}{\max(\text{nrmFua}, \text{nrmPmf}, 1)}$$

where $i \in \{0..\text{nrmPmf}\}$ and $j \in \{0..\text{nrmFua}\}$.

Similarity of Fields Written Before Another Field: We compare the similarity of fields written before another field analogously to the *Similarity of Fields Read Before Another Field*.

Similarity of Loop Reads: To distinguish fields with a collection type from those without one, we check whether a method reads a field in a loop. However, loops are not easily identified in Java bytecode because some obfuscators change control-flow graphs to hide loop structures. For this reason, *LibMapper* first calculates the strongly connected components of the control-flow graph and then checks whether the a field is read in a strongly connected component. To aggregate this value across all reading methods and all strongly connected components, *LibMapper* uses the following formula to match the loop reads of the field under analysis (lrFua) and the potentially matched field (lrPmf).

$$\text{Similarity of Loop Reads}_{ij} = 1 - \frac{|lrFua - lrPmf|}{\max(1, lrFua, lrPmf)}$$

Similarity of Loop Writes: We handle the field-writes in a loop analogously to the *Similarity of Loop Reads*.

Similarity of Assigned Entities: We use the previous field data to distinguish fields that methods access using different variations. However, we cannot distinguish obfuscated fields with the same field type in data container classes accessed via small obfuscated methods such as getters and setters. It is hard to resolve such cases because the obfuscated methods might have not enough content information to resolve the fields and vice versa.

Nevertheless, if the obfuscated methods write different values to these fields, the values can be used to distinguish the fields. For this reason, *LibMapper* compares for each method that writes a field, which entities are written into the field. These entities can be values of common types such as `java.lang.String` or `int`, but also results of method calls or field accesses. *LibMapper* uses the fuzzy signatures of methods and fields to encode these connections. To compare as few values as possible, we compare for each method only the first assigned value (av_i, av_j) for the number of methods for the field under analysis ($favFua$) and the potentially matched field ($favPmf$) using the following formulas:

$$Value\ Similarity_{ij} = equal(av_i, av_j)$$

$$Similarity\ of\ Assigned\ Entities = \frac{\sum_i \max_j(Value\ Similarity_{ij})}{\max(favPmf, 1)}$$

where $i \in \{0..favFua\}$ and $j \in \{0..favPmf\}$.

6.3.7. Match Class Data

In addition to the information from methods and fields, we need a score for potentially matched classes. Since the suggestions of *LibDetect* and our class representations only encompass the core functionalities of a class and not its surrounding code entities, we use as features the former and the latter to identify a name for the class. In the following, we describe the used class features to determine the mapping for a library class.

Fully-Qualified Name: Obfuscators cannot change all class names because different sources may reference them, and obfuscators have no access to these references. *LibMapper* uses 1.0 for a complete match of a class name and 0.0 otherwise.

Similarity of Subclasses: While an obfuscator may have changed the name of a class, it is still possible that subclasses of that class keep their original names because sources reference these names, which are not controlled by the obfuscator. *LibMapper* compares each subclass name (sc_i) of the class under analysis (cua) with the subclass names (sc_j) of a potentially matched class (pmc). For the combination of multiple subclass-names, *LibMapper* uses the following formulas:

$$Subclass\ Similarity_{ij} = equal(sc_i, sc_j)$$

6. Library Mapping

$$Similarity\ of\ Subclasses = \frac{\sum_i \max_j (Subclass\ Similarity_{ij})}{\max(\#pmc, 1)}$$

where $i \in \{0..\#pmc\}$ and $j \in \{0..\#cua\}$.

Similarity of Super Interface Classes: *LibMapper* calculates the similarity of super interface classes analogously to *Similarity of Subclasses*.

Class Signature: While matching the direct superclasses and interfaces might match some of the classes, most classes have obfuscated superclasses and interfaces. To handle this fact, *LibMapper* uses the *Nameless Representation* of the class under analysis and compare it with the NR of a potentially matched class. This comparison allows us to match classes that transitively inherit from the Android class library. As score, *LibMapper* uses 1.0 for a complete match and 0.0 otherwise.

Similarity of fan-In: As with subclasses, superclasses, and interfaces, *LibMapper* compares the classes that access the class under analysis with the classes that access the potentially matched class (acPmc). In order to calculate a score for the match, it uses the following formula:

$$Similarity\ of\ Fan\ In = \frac{\#accessing\ classes}{\#acPmc}$$

Similarity of fan-Out: *LibMapper* calculates the number of matched classes accessed by the class under analysis analogously to the *Similarity of Fan In*.

Number of Instructions: While an obfuscator might change the names of the surrounding classes, the number of instructions usually stays similar. *LibMapper* compares this number between the class under analysis (niCua) and a potentially matched class (niPmc) using the following formula:

$$Instruction\ Count\ Similarity = 1 - \frac{|niCua - niPmc|}{\max(1, niCua, niPmc)}$$

Number of Methods: We compare the number of methods analogously to the number of instructions using a similar formula for the class under analysis (nmCua) and a potentially matched class (nmPmc):

$$Method\ Count\ Similarity = 1 - \frac{|nmCua - nmPmc|}{\max(1, nmCua, nmPmc)}$$

Class Kind: Android and Java bytecode use some classes for unique purposes. For instance, some classes are used as enumerators of specific values, are abstract classes, are anonymous classes, are interfaces, inherit from Exception or Error, or are resource classes. To match these different kinds of classes, *LibMapper* compares the kind of the class under analysis with each potentially matched class. Since these class kinds are mutually exclusive, we match them using 1.0 for a full match and 0.0 otherwise.

6.3.8. Preferred Classes

After *LibMapper* resolved a class mapping, it uses its database to add all class names that access the mapped class to a collection of preferred classes. These classes are used to match others in the following iterations. A class in the preferred-class collection has a slightly higher chance to be picked in future mappings because *LibMapper* calculates an additional score for classes that match a class in the collection. *LibMapper* represents the class names of the collection in our resolving process, as a score: 1.0 for a full match and 0.0 for no match.

6.3.9. Resolve Mapping

Given the mapped method-, field-, and class data for all potentially matched classes, the algorithm finally maps one of the potentially matched classes to the class under analysis. For this, it averages all scores of the mappings and chooses the maximum-fitting class-mapping. After this step, each analyzed class was mapped to a single library class with the previously assigned method and field mappings. In the end, the algorithm outputs the mapping of all classes into a file.

After describing our mapping procedure, we evaluate the effectiveness of *LibMapper* against different obfuscation configurations and the state-of-the-art deobfuscator *DeGuard* in the next section.

6.4. Evaluation

In this section, we evaluate our *LibMapper* against the state-of-the-art approach *DeGuard*. Furthermore, we analyze *LibMapper*'s false positives and false negatives. We chose *DeGuard* because it is the most advanced tool for resolving obfuscated names in Android apps. For the evaluation, we analyzed *LibMapper*'s effectiveness and compared it with *DeGuard* by investigating the following research questions:

RQ1: How effective is *LibMapper* compared against *DeGuard* to map names of obfuscated libraries in the standard obfuscator configuration?

RQ2: How effective is *LibMapper* compared against *DeGuard* to map names of obfuscated libraries in an advanced obfuscator configuration?

RQ3: What kind of entities are mapped incorrectly by the *LibMapper*?

For the experiments, we used a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The analyses were executed using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory and MySQL 5.7 for the library database.

To answer these questions, we used all 453 apps from the case study in Section 6.2, which do not contain obfuscated names. While we stored the libraries from 302 apps in *LibMapper*'s database, we used the remaining 151 apps for the comparison with *DeGuard*. For the comparison, we obfuscated the 151 apps using *ProGuard* and used

6. Library Mapping

the mapping files which *ProGuard* created to assess the effectiveness of *LibMapper* and *DeGuard*. To this end, we compared each tool’s precision, recall, and harmonic mean of both values (F_1 -measure). For calculating the precision, recall, and F_1 -measure, we checked each entry in the mapping file that was created by *ProGuard* with a corresponding entry in the deobfuscation mapping files from *LibMapper* or *DeGuard*. If an entry belongs to a library and the deobfuscated name completely conforms with the original name, we count it as a true positive. However, if the entry in the original mapping does not belong to a library and the deobfuscation mapping assigns this entry a library name, we count it as a false positive. Analogously, we identify false negatives and true negatives by checking whether the assigned name and the original name belong to the app code or the library code.

Using this procedure, we can only identify complete matches. However, to identify partial matches, we split the entry names using Samurai [EHPVS09] and use the fraction of matched words as a matching score.

Given the description of precision, recall, and F_1 -measure, we evaluate *LibMapper* and *DeGuard* using two configurations of *ProGuard* [Pro17]. In the following, we describe the standard configuration and the most advanced configuration for name obfuscation. Many developers use the former to build their app, but the latter could provide more protection for the developer’s intellectual property.

6.4.1. Standard Name Obfuscation Configuration

We use the standard configuration of *ProGuard* [Pro17] to obfuscate the names of the 151 apps and compare the abilities of *LibMapper* and *DeGuard* to resolve the obfuscated names. We use *ProGuard* because it is integrated into the build system of Android apps and, therefore, often used in practice. With *ProGuard*’s standard configuration, the tool obfuscates names by replacing meaningful names with meaningless short character sequences. For instance, the method name `getLength` may be obfuscated to the name `a` or any other combination of characters.

Given the obfuscated apps, we execute *LibMapper* and *DeGuard* on them to measure the precision, recall, and the F_1 -score, which we show in Figure 6.3.

Observation 10 While *DeGuard* has a 0.79% higher precision (99.92% vs. 99.14%), *LibMapper* has a 5.1% higher recall (88.12% vs. 83.84%). The high recall results in a higher F_1 -score of *LibMapper* (93.31% vs. 91.18%). Therefore, in the standard configuration of *ProGuard*, *LibMapper* resolves more obfuscated names than *DeGuard* (**RQ1**).

6.4.2. Advanced Name Obfuscation Configuration

For the second experiment, we enabled the repackaging option of *ProGuard* and obfuscated the names by words from a natural language dictionary [Sta20]. The repackaging causes the relocation of all classes into the root package of an app. While this option

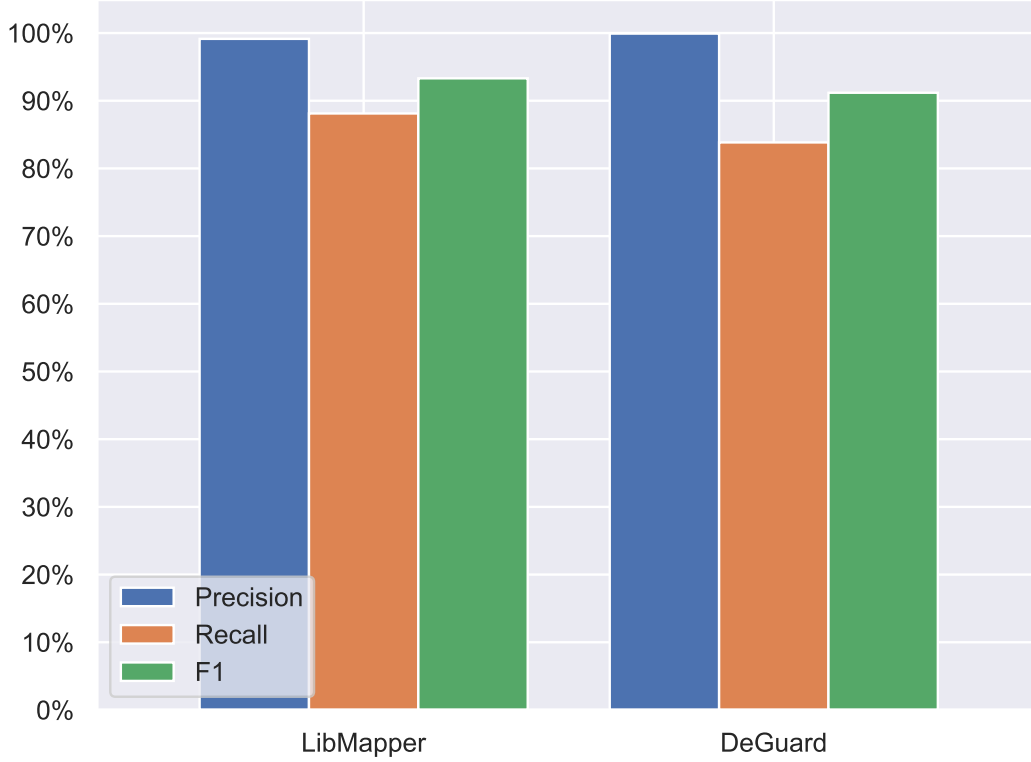


Figure 6.3.: Average precision, recall and F1-measure of *LibMapper* and *DeGuard*

does not directly affect the class names, it prohibits deobfuscators from keeping the same simple names in the same package without renaming the simple names.

The usage of a natural language dictionary circumvents deobfuscators that only resolve names with meaningless character sequences. For instance, the method name `getLength` is changed to `John` which is not just a sequence of random characters like `abc`.

Using these two options, we obfuscated the names of the 151 apps and resolved the library names using *LibMapper* and *DeGuard*. Figure 6.4 shows the comparison between *LibMapper*'s and *DeGuard*'s average precision, recall, and F_1 -score for the apps obfuscated using the above-described advanced configuration.

Observation 11 *While in this experiment, DeGuard has a higher precision than LibMapper (64.39% vs. 83.47%), LibMapper achieves more than eight times the recall of DeGuard (6.64% vs. 54.41%). These differences result in a significant distance between the harmonic means of both approaches (58.98 vs. 12.3%). While the difference in precision and recall is higher than in the first experiment, both tools map less code entities correctly. However, LibMapper resolves obfuscated names better than DeGuard in the advanced name-obfuscation configuration of ProGuard (RQ2).*

6. Library Mapping

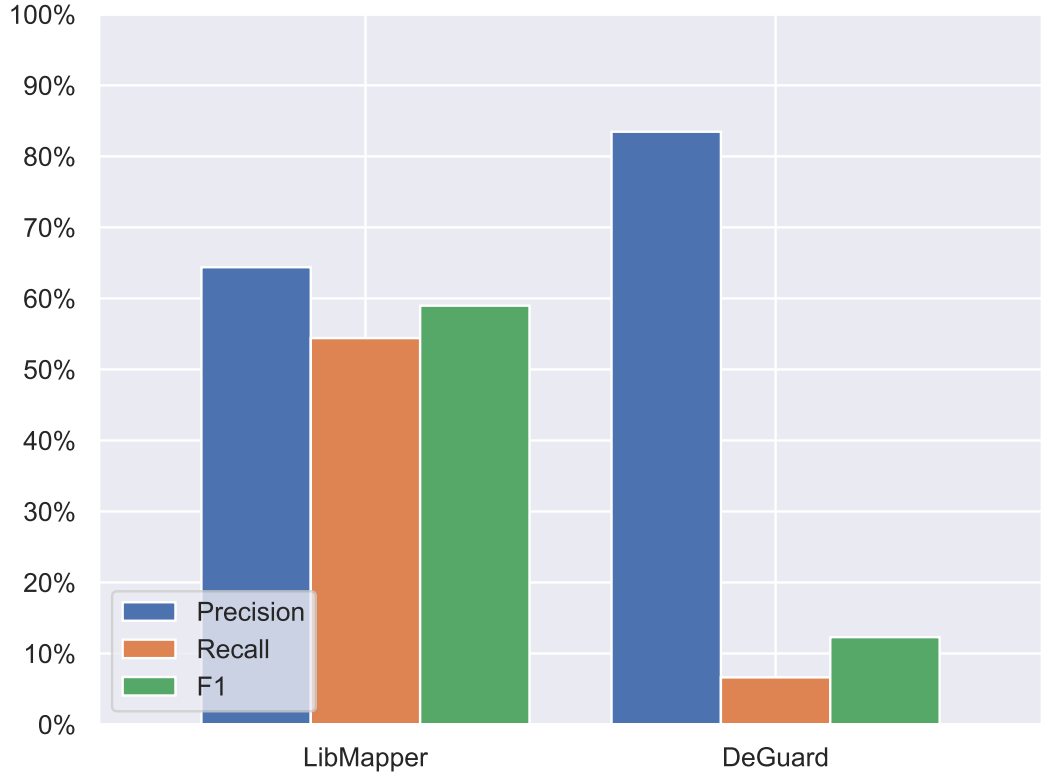


Figure 6.4.: Average precision, recall and F1-measure of *LibMapper* and *DeGuard* for the advanced configuration

6.4.3. False Negatives of LibMapper

To evaluate the causes of *LibMapper*'s false mappings, we analyzed false negatives using the obfuscation from *ProGuard*'s configurations and the deobfuscation mappings from *LibMapper*. We consider as false-negatives library entries that either did not match at all or only partially. Using the identified false negatives, we examined fully-qualified names of fields, methods, and classes.

Figure 6.5 shows the missing names for each obfuscator configuration. Using the standard configuration causes a total of 11.88% false negatives, consisting of 8.89% false field names, 2.6% false method names, and only 0.39% false class names. We randomly chose 100 fields and examined their context. Most of the fields with false name mappings are rarely used and do not have any distinctive features.

Using the advanced configuration causes more than 28 times more incorrectly assigned class names because of classes being repackaged into the root package of an app. With the repackaging, less information is available to map fields and methods, which results in an amplification of the effect mentioned earlier for fields. Since fewer classes can be mapped, 23.98% of the field names are falsely assigned. With fewer mapped classes and fields, 10.28% of the method names are also falsely assigned.

Observation 12 Due to these results, we identify for **RQ3** two main causes of false negatives: The first cause is fields that are only used by less descriptive methods, such as `get*` or `set*` methods and that have a widespread field type. The second cause is the repackaging of classes, which results in the fact that small classes cannot be mapped unambiguously.

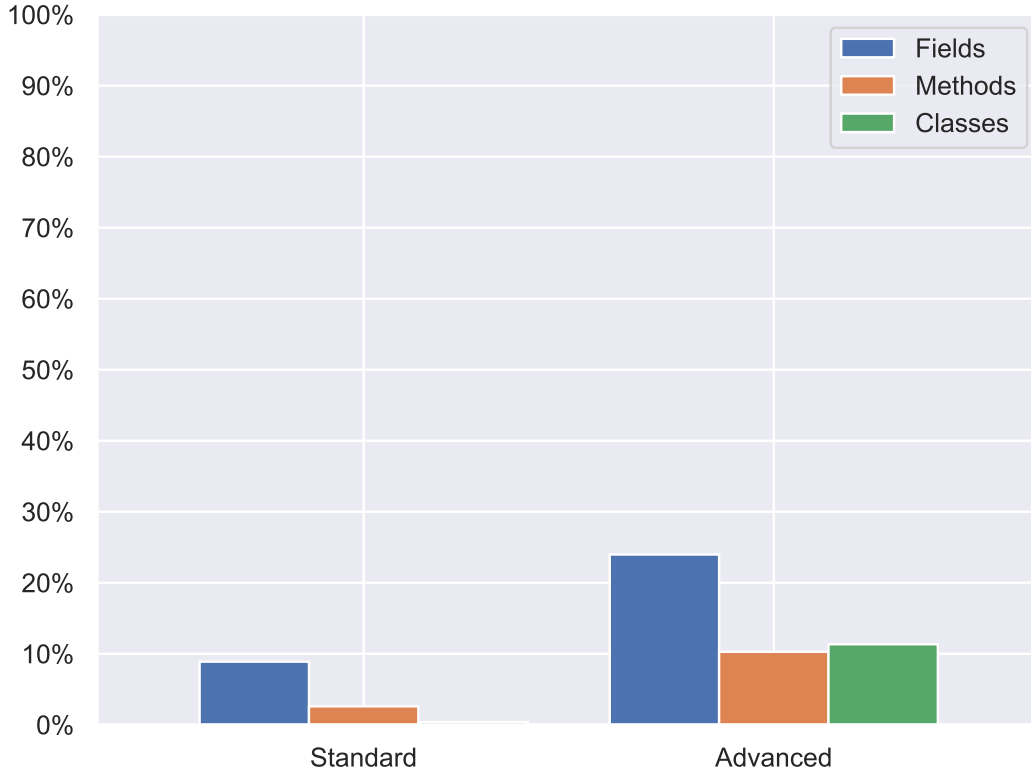


Figure 6.5.: Identified false negatives per configuration categorized by field, method, and class names.

6.4.4. Discussion

In the experiment with *ProGuard*'s standard configuration, *DeGuard*'s recall is already 5% lower than *LibMapper*'s but *DeGuard*'s recall drops to 6.64% when using the advanced configuration. To identify the cause of *DeGuard*'s low recall, we manually compared its output files with the mapping files from *ProGuard*. The comparison showed that *DeGuard* changed only 6.64% of the names that *ProGuard* obfuscated using the given dictionary. This fact suggests that it is not a false assignment that causes the low recall but a false identification of non-obfuscated code entities.

In the experiment with the advanced configuration, *LibMapper* has a lower precision than *DeGuard*, which is caused by repackaging all classes into the same package. This

6. Library Mapping

repackaging blurs the boundaries between library and app code and causes app classes to be considered as library classes. Because *DeGuard* identifies only a small fraction of the obfuscated names, it changes only those names and therefore has higher precision. However, since many obfuscated names were not identified by *DeGuard*, it has a low recall. In contrast, if *LibMapper* considers an app class as a library class, it also considers all its members belonging to a library class, which leads to the lower precision. The low precision can be improved by improving *AppSeparator*'s splitting procedure.

The missing library mappings in *LibMapper*'s output files are also caused by the repackaging so that library classes are considered as app classes and, therefore, produce many false negatives. However, in the experiment with the advanced configuration, *LibMapper*'s recall is significantly higher than *DeGuard*'s. Therefore, it is much better suited to map names to obfuscated library code entities.

6.5. Threats to Validity

In the following, we discuss threats to the validity of our library mapping experiments.

Since *LibMapper* maps only code that was identified as library code by *AppSeparator*, missing entries in *AppSeparator*'s reference white list may lead to missing library mapping. However, due to the high precision of *AppSeparator* even without the white list, the overall error should be negligible.

Assembling an extensive, up-to-date ground truth database to learn the mapping of library code is infeasible because some libraries contain obfuscated names. However, using further analyses that identify obfuscated names, we could extend our database with non-obfuscated library code entities.

The 151 evaluated apps from Section 6.4 may not be sufficient to compare *LibMapper* with *DeGuard*. Since we chose these apps from the same source as the database of *DeGuard*, we are confident that the results are comparable.

To evaluate the impact of obfuscation on library mapping, we used the obfuscator *ProGuard*, which applies renaming, optimization, and shrinking. To the best of our knowledge, other obfuscators either use the same renaming technique as *ProGuard* or use *ProGuard* as a post-processing step. However, other obfuscators might apply stronger techniques, such as virtualization, class encryption, class loading, and packaging. To the best of our knowledge, no existing library mapping approach handles these techniques.

6.6. Conclusion

We presented our approach *LibMapper* that uses app-code separation and library detection to collect possible candidates for library classes. Afterward, it extracts multiple features to unambiguously map each library class, field, and method with a name from non-obfuscated library code entities.

The evaluation shows that *LibMapper* maps library code better than the state-of-the-art deobfuscator *DeGuard*. Obfuscating names with *ProGuard*'s most advanced naming configuration causes that *LibMapper*'s recall is more than seven times higher than the

one of *DeGuard*. During an analysis of *DeGuard*'s mappings, we identified that most of the names are not falsely assigned but not handled at all. We suspect that an incorrect detection of obfuscated names causes this fact.

7. Detection of Obfuscated Names

Previous research [YFM⁺17, IWAM17, STDA⁺17, and17, LLB⁺17] analyzed repackaging and malicious components by extracting features from app code that indicate repackaged or malicious behavior. To identify suitable features for their approaches, researchers investigated samples of known repackaged apps and malware payloads. For the investigation, an app needs to be reverse-engineered to reveal the app’s code. In order to understand the content of the code, it needs to be comprehensible and have an easily readable layout. In previous work [PHD11, GS10, HØ09], researchers studied the code layout and found that manual analysis is more difficult without understandable names of code entities such as classes, fields, and methods. However, malware writers and repackagers use name obfuscation to hide the behavior of their apps. For that purpose, some approaches [BRTV16, RVK15, VCD17, Jaf17, AZLY19, CFPK20] support analysts by suggesting names for code entities with obfuscated names based on the assumption that in a similar context, many names are the same [GS10]. Since these approaches infer names from a code entity’s context, we refer to them as name inference approaches.

These name inference approaches have two main drawbacks. First, they do not identify which code entities have obfuscated names leading to the overwriting of non-obfuscated names. Second, the approaches do not separate between library code and app code, leading to the suggestion of names from library code to app code entities. While the previous chapter described how to handle the second drawback, we propose, in this chapter, an approach to identify obfuscated names of code entities in Android apps.

Previous works [WHA⁺18, Mir18, WR17, Don18] proposed several tools to identify apps that contain obfuscated names. However, they only measure whether an app contains obfuscated names but cannot pinpoint them in the code. This fact results from the features used by these approaches to identify obfuscated names. They either combine all names of an obfuscated app or a specific area without focusing on single names. Furthermore, all approaches target only specific name-obfuscation techniques (e.g., *ProGuard*’s standard configuration [Pro17]) so that they may miss obfuscated names produced by more advanced techniques.

We introduce *ObfusSpot* that pinpoints obfuscated names in a given codebase to support name inference approaches. *ObfusSpot* identifies obfuscated names in three steps. In the first step, it uses a classifier to identify names that were manipulated using the standard configuration of obfuscators such as *ProGuard* [Pro17], *Allatori* [All19], *DashO* [Das19], *ZKM* [Zel19], and *Stringer* [str19]. This step identifies all obfuscated names with a different pattern of character sequences than non-obfuscated names.

In the second step, *ObfusSpot* identifies obfuscated names by measuring the frequency of code entity names because non-obfuscated names are less frequent than obfuscated

7. Detection of Obfuscated Names

ones. For this analysis, it uses a trained classifier to decide which frequencies are an indication for obfuscated names.

While the second step identifies obfuscated names that occur more frequently than other names, it is laborious to identify less common obfuscated names. For this purpose, *ObfusSpot* uses *LibMapper* to check whether the names of libraries are obfuscated. If an obfuscator manipulated the names of library code entities, *ObfusSpot* extracts the used name-obfuscation pattern and tries to identify other names with the same naming pattern in the remaining code. For instance, if a technique frequently uses the word "demo" in their obfuscated names, we identify this word in names of library code entities and can flag all names of app code entities that also contain this word. Using the last two steps, *ObfusSpot* can identify obfuscated names that are rarely used or are combined from more frequent ones. To the best of our knowledge, *ObfusSpot* is the first detector that can handle obfuscated names that consist of concatenated words from dictionaries.

For the evaluation of *ObfusSpot*, we downloaded mapping files from 948 app repositories, which are used by developers to map obfuscated names back to the original names if they receive a crash report by an app user. With the knowledge from these mapping files, we built a ground truth of obfuscated and non-obfuscated names and used it to train our classifiers and measure *ObfusSpot*'s effectiveness. The results show that each of *ObfusSpot*'s steps effectively identifies names obfuscate using different techniques.

Furthermore, because the developers of obfuscators are experts for their tools and most likely obfuscated them with more advanced techniques, we measure whether *ObfusSpot* recognizes manipulated names contained in these codebases. *ObfusSpot* identifies with a F_1 -measure of 94.46% all obfuscated names in the codebases of obfuscation tools.

Finally, we analyzed 100,000 apps to measure the percentage of apps that were obfuscated by their developers, libraries that are only available in obfuscated form, and apps that are obfuscated using a dictionary instead of random sequences of characters. The results show that only 29.33% of the analyzed apps were obfuscated by their developers, and only 3.59% of the libraries are released in obfuscated form. Interestingly, the names of 18.53% of the obfuscated code entities could only be found using *ObfusSpot*'s last two steps. These names were obfuscated using natural language dictionaries.

In the next section, we discuss the related work (Section 7.1) followed by the description of our approach *ObfusSpot* in Section 7.2. We evaluated the approach in Section 7.3, discuss in Section 7.4 threats to the validity of our experiments, and draw a conclusion of our work in Section 7.5.

7.1. State-of-the-Art Detectors of Obfuscated Names

In this section, we describe related approaches to identify apps that contain obfuscated names. All approaches use either a classifier or heuristics to identify names manipulated by a specific obfuscator.

Dong et al. [Don18] identify apps with obfuscated names by counting the occurrence of 3-grams in each name of a code entity and using these counts to train a Support

Vector Machine (SVM) classifier [Wan05]. This SVM distinguishes between apps with obfuscated names and non-obfuscated names. While this approach identifies whether an app contains obfuscated names, it cannot pinpoint an individual obfuscated name because it operates on all names of an obfuscated app.

AndrODet [Mir18] identifies apps with obfuscated names using seven classifiers. These classifiers are trained with multiple aggregated features, and their classifications are combined using limited random search [Pri77]. Each newly analyzed app is used by *AndrODet* to improve its models. However, *AndrODet* only identifies if entire apps are obfuscated and cannot pinpoint the exact name of a code entity because it aggregates its features using all names in an app.

Wang et al. [WR17] identify apps with obfuscated names by determining the used obfuscator and even the used configuration. For this purpose, their approach extracts all names and strings, filters them, counts their occurrence, and uses the occurrences to train an SVM for each configuration of known obfuscators. The resulting SVMs can identify which name obfuscation was used to manipulate an app. Unlike *ObfusSpot*, the described tool only identifies if an app contains obfuscated names but cannot pinpoint which of the names are obfuscated.

Park et al. [PYC⁺19] identify apps with obfuscated names by decompiling all Java classes from an app, treating the classes as text documents, and classifying the documents using various machine learning techniques. The approach can identify different obfuscation techniques applied to all classes of an app, such as renaming, string obfuscation, control-flow obfuscation, and the usage of reflection. However, it cannot pinpoint the exact location of an obfuscated name because it uses all classes to identify the various obfuscation techniques.

Kaur et al. [KNGS18] identify obfuscated names by transforming the bytes of an app's codebase into an image and using image classification to determine the used obfuscation technique. As the approaches of Park et al. [PYC⁺19] and Wang et al. [WR17], this approach can identify the different obfuscation techniques used to manipulate an app. However, this approach is also not able to pinpoint the exact location of an obfuscated name.

OBFUSCAN [WHA⁺18] identifies obfuscated names of packages, classes, fields, and methods by using heuristics extracted from analyzing the implementation of the most used obfuscator *ProGuard* [Pro17]. Additionally, it checks whether an obfuscator removed the debug information, the source code's reference, and specific annotations. The authors of *OBFUSCAN* used it for a large scale study of 1,762,868 apps to measure the impact of all these obfuscation techniques and how often the main code of an app is obfuscated using these techniques. However, *OBFUSCAN* focuses only on names obfuscated

7. Detection of Obfuscated Names

by *ProGuard* but cannot identify obfuscated names manipulated by other obfuscators. Additionally, it does not pinpoint the exact location of an obfuscated name.

Discussion Some approaches identify different obfuscation techniques or even the used obfuscator configuration using either machine learning or heuristics. However, none of the approaches analyzed name obfuscation using a natural language dictionary. While the standard configurations of obfuscators manipulate names using random sequences of characters, names obfuscated using a natural language dictionary are hard to distinguish from non-obfuscated names. Only *ObfusSpot* can identify such names by using the frequency of names and detecting patterns of obfuscated names.

Additionally, most approaches identify names obfuscated using standard configurations by training their classifiers with obfuscated and non-obfuscated names. In contrast, *ObfusSpot* uses only non-obfuscated names to detect anomalous characteristics of obfuscated names. Finally, unlike *ObfusSpot*, all other approaches can only detect whether the names in an app are obfuscated but cannot pinpoint the exact locations of such names.

7.2. ObfusSpot

In this section, we describe our approach *ObfusSpot* that identifies obfuscated names in three steps. The first step combines a clustering and a classification algorithm to identify anomalies in the naming scheme of obfuscated names. This step is similar to the approach of Gadai et al. [GM17] that focuses on intrusion detection. The second step uses a classifier to identify names of obfuscated code entities by using the frequency of the names as a feature because non-obfuscated names are less frequent than obfuscated ones. As last step, *ObfusSpot* uses *LibMapper* (cf. Section 6.3) to extract naming patterns of obfuscated library code entities. Using these patterns, *ObfusSpot* identifies obfuscated names of app code entities that are obfuscated similarly.

In the following, we describe *ObfusSpot*'s processing steps depicted in Figure 7.1. First, *ObfusSpot* analyzes the code using anomaly classification and frequency classification. If the second step identifies other obfuscated names than the first step, *ObfusSpot* use *LibMapper* to identify obfuscated names in libraries. Afterward, the Name Matcher identifies all names of app code entities that use the same naming scheme as the identified obfuscated names and outputs the results to the user.

While *ObfusSpot* always performs the anomaly and frequency classification, it executes the library mapping only if the frequency classification identified more obfuscated names than the anomaly classification. *ObfusSpot* uses this condition to ensure that it executes the time-consuming library mapping only if the app contains more obfuscated names than can be found with the first step.

The following describes the processing steps and the interaction between the different processing steps in more detail.

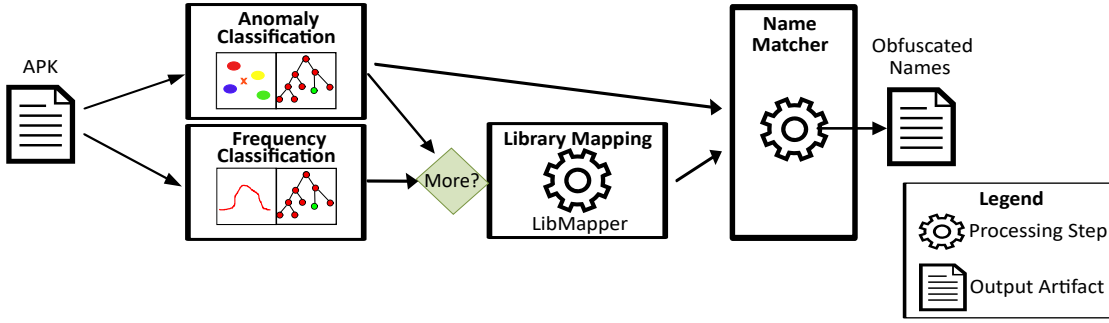


Figure 7.1.: Process to identify obfuscated names

7.2.1. Anomaly Classification

For the detection of anomalies in names of obfuscated code entities, *ObfusSpot* extracts several features shown in Table 7.1. First, it clusters all names and then identifies all names that do not fit into these clusters. The intuition behind the features is that non-obfuscated names contain words, are not encrypted or compressed, and have a well defined distribution of vocals, consonants and other characters. In contrast, names changed by an obfuscator can be encrypted or compressed (e.g., `getLength` → `a2f34d`), can be transformed into a random sequences of characters (e.g., `getLength` → `IiIiiiIiI`), or just minified to a single character (e.g., `getLength` → `a`).

In the following, we describe the different features from Table 7.1 by their category.

Statistical Tests: Previous statistical analyses of encryption mechanisms [Kah96, cry19, LH07] show that obfuscated names often have a random (close to equal) distribution of characters. Consequently, *ObfusSpot* identifies a random distribution and uses it as a discriminating feature to distinguish between obfuscated and other names with special characters. Thereby, *ObfusSpot* uses three different measures to check whether the distribution of the characters is random because we encountered that each one is suited for different scenarios. For instance, previous works [LH07] used the *Normalized entropy* to identify encrypted malware. *ObfusSpot* uses it to identify encrypted names. While the *Chi-squared* test measures the deviation of the characters from the equal distribution, the *average distribution* measures whether an obfuscator rotated the characters of a given name (e.g. caesar cipher [Kah96]) or it belongs to a language.

Compression Rate: If *ObfusSpot* uses only the statistical tests, it may miss names obfuscated using compression, e.g., compressing a name using *gzip*. To correctly identify such names, *ObfusSpot* compresses all names and compares the compressed name’s length against the original name length. Based on cryptanalysis knowledge [cry19], the comparison will show that both lengths do not differ if the original name already consisted of compressed data.

Word Counts: As described above, names changed with the standard configuration

7. Detection of Obfuscated Names

of an obfuscator contain no or just a few words. *ObfusSpot* uses natural language dictionaries to check whether a name contains words. However, some names cannot be checked because they do not follow the camel-case naming convention [ora19]. As a result, *ObfusSpot* splits names from non-Latin languages using ICU-Tokenizer [ICU19] and *Samurai* [EHPVS09]. *Samurai* splits identifiers by a list of frequently used words.

Name Characteristics: *ObfusSpot* uses one feature from *AndroDet* [Mir18] that identifies if an app uses name obfuscation. While *AndroDet* aggregates all name features over the app, *ObfusSpot* uses it to classify individual names. Additionally, to the feature from *AndroDet*, *ObfusSpot* combines eight additional ones that calculate different character distributions, e.g., character counts, digits.

Table 7.1.: Feature list for the clustering of obfuscated names

Category	Name	Description
Statistical Tests	Chi-squared Test	Tests if all chars in the given name are equally, distributed indicating a random distribution.
	Average Distribution	The average distribution which is close to the Gaussian distribution for plain names,
	Normalized Entropy	The normalized entropy of names.
Compression Rate	GZIP	The rate of the GZIP compression,
Word Counts	Dictionaries of 54 Languages [Mil95, dic20]	The shortest word length, the largest word length, the number of words, the number of unique words from a multiple language dictionary.
Name Characteristics	AndroDet[Mir18]	Sum of repetitive characters.
	Character Counts	Number of characters, Number of vocals, Number of consonants, Number of digits, Number of unique characters, Number of non letters, Maximum number of consecutive characters, Maximum occurrences of the same character

Training Given the 17 features in Table 7.1, *ObfusSpot* uses a data set with non-obfuscated names to create clusters from the features that represent the non-obfuscated names. Afterward, it uses the expectation-maximization (EM) algorithm (cf. Section 2.3.1) from Weka [WF02] to build clusters of these names.

After the extraction of clusters, we use a data set of obfuscated and non-obfuscated names for the training of our classifier. From this data set, we measure for each data entry the Euclidean distance to each cluster center. For the measurement, we standardize the extracted feature values using the following formula: $\frac{x_i - \text{mean}_i}{\text{std}_i}$ where the mean and standard deviation (std) is used from each cluster.

Given the distances to all cluster centers, we use the minimum distance (mindistance) to a cluster center and the cluster name (mincluster) to train a binary classifier that identifies name anomalies. However, before we use these values, we exclude outliers and extreme values using Weka’s interquartile ranges. The intuition behind these features is that obfuscated names have a higher distance to specific cluster centers than non-obfuscated names. Since we cannot anticipate the optimal distances and cluster values for these features, we train multiple classifiers to evaluate the most suitable classifier that identifies these values. The evaluation of the most suitable classifier is discussed in Section 7.3.1.

Classification For the classification, we extract the mindistance and mincluster values as in the previously-described process and use the trained binary classifier to identify obfuscated names.

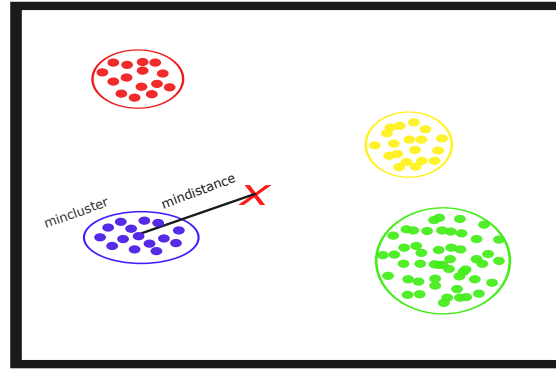


Figure 7.2.: Cluster example

Let us assume that the cluster centers are represented in two-dimensional space like the colored ovals in Figure 7.2, and the features of an obfuscated name are at the point of the red X. We can see that X is closest to the blue oval, which is our **mincluster**, and we take the Euclidean distance from this oval for our classifications as **mindistance**. Since X is too far away from the **mincluster**, the name that represents the X is classified as obfuscated.

7.2.2. Frequency Classification

In the second step, *ObfusSpot* extracts the occurrence of each code entity name in an app. For this purpose, *ObfusSpot* counts for each name how often it occurs in a simple name of a class, in a field name, and a method name (except names of initiators such as constructors or static initializers).

After calculating the occurrences, *ObfusSpot* measures each name’s distribution using the number of all names with the same frequency (nnf) and the frequency of the name under analysis (nf). Given these values, it computes the name distribution with the following formula: $\frac{nf}{\max(1, nnf * nf)}$ This feature’s intuition comes from the insight that

7. Detection of Obfuscated Names

obfuscators overload many names. The overloading creates a higher distribution of the same names in an app, and *ObfusSpot* exploits this insight to identify obfuscated names.

A name distribution closer to 1.0 indicates that this name occurs more frequently in comparison to other names. In contrast, a value closer to 0.0 indicates that the name under analysis occurs as frequently as other names. However, some non-obfuscated names may have a similar distribution to obfuscated names and vice versa. Since we cannot determine the optimal distribution point to decide whether a name is obfuscated, we train a classifier that learns the closest representation to this point from a given data set of obfuscated and non-obfuscated names. The trained classifier outputs a binary decision for each name using only the name distribution and its frequency.

7.2.3. Library Mapping

In the last step, *ObfusSpot* uses library mapping to identify obfuscated names. However, *ObfusSpot* executes this step only if it encounters other obfuscated names in the output of the second step than in the one of the first step. Because the identification of other names in the output of the second step might indicate that an obfuscator changed the names of an app using a natural language dictionary.

For instance, if an obfuscator manipulates the code entities' names in an app with random character sequences, then the anomaly classification and frequency classification would identify the same obfuscated names because these names are anomalous and very frequent. However, if the obfuscator uses random character sequences and words from a natural language dictionary, then the first step would identify other names than the second step. The same applies if an obfuscator only uses the natural language dictionary because the first step would identify no obfuscated names, while the second step might identify many obfuscated names with a higher distribution.

Since some names obfuscated with the natural language dictionary have a low distribution, we use *LibMapper* from Section 6.3 for the library mapping. If *LibMapper* assigns to a code entity another name than it had before, we consider the previous name as obfuscated.

7.2.4. Name Matcher

After the three steps' execution, *ObfusSpot* splits each obfuscated name using Samurai [EHPVS09] to identify whether other names use the same words as the obfuscated ones. If a previously not identified name consists entirely of words from obfuscated names, it is also considered obfuscated. With this procedure, *ObfusSpot* identifies obfuscated names that are concatenated from obfuscated words found in names of library code entities of an app. In the end, *ObfusSpot* outputs a list of obfuscated class-, field-, and method names.

Given the description of *ObfusSpot*, we evaluate, in the next section, its effectiveness against obfuscated names in the wild.

7.3. Evaluation

For the evaluation of *ObfusSpot*'s effectiveness, we conducted five experiments to answer the following research questions:

RQ1: What are the best classifiers for *ObfusSpot* to identify obfuscated names?

RQ2: How accurate is *ObfusSpot* in finding obfuscated names?

RQ3: How effective is *ObfusSpot*'s identification for names obfuscated by experts?

RQ4: How many apps contain obfuscated main code?

RQ5: How common are libraries released only in obfuscated form?

We performed our experiments using a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The experiments were executed on the server using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory for Weka 3.9.4.

For our evaluation, we downloaded and analyzed four different data sets. The first data set consists of all 453 apps that we have described in Section 6.2.1. These apps are the only ones from F-Droid [F-D19] that contained no obfuscated names. The second data set consists of 948 mapping files extracted from GitHub [git20] using a BOA script (cf. Appendix Listing 12.1). Developers use these files to map the obfuscated names back to original names if they receive a crash report from their app users. The third data set consists of seven obfuscator trial-versions. It was downloaded to show how often these obfuscators contain obfuscated names. The last data set encloses 100,000 apps randomly selected from AndroidZoo [HAB⁺16].

7.3.1. Selection of Classifiers

To classify obfuscated names, we first use the 453 apps from the F-Droid store to extract all non-obfuscated names for the clustering step of the anomaly classification (cf. Section 7.2.1). In total, we extracted 11,338,654 names from classes, fields, and methods. Using these names, we calculated our features and clustered them with the expectation-maximization (EM) algorithm from Weka [WF02]. The EM algorithm outputted five clusters with the means and standard deviations for each feature.

Anomaly Classification For the classification, we used the 948 mapping files downloaded from app repositories of GitHub. With these mapping files, we can deduce which names an obfuscator changed in an app. If the output name is not equal to the input name, the output name is flagged as obfuscated. However, if the input name equals the output name, we manually checked whether the names are really non-obfuscated because an obfuscator could manipulate the output name in the same form as the input name.

In total, we extracted from the 948 mapping files 3,359,533 names, of which 1,710,654 were obfuscated. To get an equal split between obfuscated and non-obfuscated names,

7. Detection of Obfuscated Names

we chose all obfuscated names and used a random sample of the same number of non-obfuscated names. Afterward, we used 80% of the resulting names for training and testing and 20% for validation.

To identify the best classifier, we measured the Euclidean distance for each name in the training and testing set and removed all outliers and extreme values. Afterward, we compared Naive Bayes, Multilayer Perceptron, Logistic Regression, Random Tree, Random Forest, and REP Tree from Weka’s [WF02] collection to find the best classifier for our approach. While most classifiers were trained using their default values, the Multilayer Perceptron was evaluated using three different epoch values because of their impact on the classifier’s performance.

The eight classifiers were trained and tested using 10-fold cross-validation [HTF09] on the names of the training set. Given the cross-validation statistics, we compare the correctly-classified instances and the root mean squared error of the classifiers. This value is a measure of false classifications of a model.

Table 7.2.: Classifier selection for the anomaly classification

Classifier	Correctly Classified	Root Mean Squared Error
Naive Bayes	97.43%	15.37%
Multilayer Perceptron (one epoch)	99.06%	9.17%
Multilayer Perceptron (five epochs)	99.06%	8.87%
Multilayer Perceptron (10 epochs)	99.06%	8.81%
Logistic Regression	99.06%	9.74%
Random Tree	99.19%	8.24%
Random Forest	99.19%	8.24%
REP Tree	99.19%	8.26%

Table 7.2 shows the correctly classified and falsely classified measures of each of the eight classifiers. Naive Bayes, with 97.43%, has the worst percentage of correctly classified instances. However, the percentage of correctly classified instances of all Multilayer Perceptron cells and the Logistic Regression is only 1.63% points higher than those of Naive Bayes. Nevertheless, the Random Tree, Random Forest, and REP Tree have the highest number of correctly classified instances (99.19%). While the Random Tree and Random Forest have the lowest false classification rate, we can integrate the REP Tree model directly into our approach, eliminating the time required to load the classifier.

Observation 13 *Since all classifiers have only a slight difference in their performance, we chose REP Tree as our preferred classifier, because of the eliminated loading overhead mentioned in the Section 2.3.2.*

Frequency Classification For the classifier selection for the frequency classification, we used the 948 mapping files and divided them into 80% (758 files) training and test set and 20% (190 files) validation set. For the classifier’s construction, we extracted each name of the mapping files and calculated its usage frequency to derive the features

described in Section 7.2.2. From the previous experiment, we already know all obfuscated names in the mapping files and used the same settings for the evaluation of this one.

Table 7.3.: Classifier selection for the frequency classification

Classifier	Correctly Classified	Root Mean Squared Error
Naive Bayes	88.24%	33.59%
Multilayer Perceptron (one epoch)	88.24%	31.15%
Multilayer Perceptron (five epochs)	88.24%	31.13%
Multilayer Perceptron (10 epochs)	88.24%	31.13%
Logistic Regression	88.20%	31.40%
Random Tree	91.33%	24.98%
Random Forest	91.33%	24.98%
REP Tree	91.33%	25.66%

Table 7.3 shows the results for each classifier. While the Logistic Regression returns the lowest percentage of correctly classified instances (88.2%), Naive Bayes and all Multilayer Perceptron configuration yield only a slightly higher percentage (88.24%). All tree-based classifiers have the highest percentage of correctly classified instances, but Random Tree and Random Forest have a lower false classification rate (24.98% vs. 25.66%).

Observation 14 *As for the previous experiment, all classifiers have only a slight difference in their performance; therefore, we use for this step also the REP Tree as our preferred classifier. Taking all experiments into account, REP Tree represents the best classifier for both data sets (RQ1).*

7.3.2. Effectiveness of *ObfusSpot*

Given both classifiers, we measure the effectiveness of both classifications using their respective validation sets. Additionally, we analyze names from the codebases of obfuscators and names that were obfuscated using dictionaries to evaluate the effectiveness of *ObfusSpot*'s steps individually.

For the evaluation of the classifiers, we use the remaining validation sets from the previous experiment. While the first validation set contains 684,262 names equally split into obfuscated and non-obfuscated names, the second one contains a total of 190 mapping files for all apps. In order to evaluate *ObfusSpot*, we extracted the features for each name in the data sets and executed the respective classifiers on them.

Table 7.4 shows the results from the execution of both classifiers on their respective validation sets. Since we extracted both validation sets from the same sources, the overall measures should be comparable. In the first classification experiment, the classifier identified about 8.68% more obfuscated names than in the second one (recall: 99.99% vs. 92%), and the precision was also 8.25% higher than in the second experiment. These results suggest that the detector of the anomaly classification would be sufficient to identify obfuscated names. However, in contrast to the anomaly-detection features, we can use the features from the second classifier if an obfuscator uses names from a natural

7. Detection of Obfuscated Names

Table 7.4.: Precision, recall, and F_1 -measures for anomaly classification and frequency classification

Measures	Anomaly Classification	Frequency Classification
Precision	98.29%	90.80%
Recall	99.99%	92.00%
F_1	99.13%	91.40%

language dictionary. Therefore, we need both classifiers to identify as many obfuscated names as possible.

Evaluation of Name Obfuscation in the Codebase of Obfuscators To measure the effectiveness of *ObfusSpot* on name obfuscation performed by experts, we analyzed the names of entities in the codebases of obfuscators. While name obfuscation performed by users may accidentally miss obfuscating all names, the developers of obfuscators known their product and use more advanced name obfuscation techniques. For this analysis, we acquired trial versions of Allatori [All19], DashO [Das19], DexGuard [Dex19c], Jarg [JAR20], Stringer [str19], yGuard [yGu20], and ZKM [Zel19] and analyzed manually whether the names of the tools’ code entities are obfuscated. Instead of analyzing every single name of simple names of classes, fields, and methods, we analyzed only the unique names, which reduced the number of names by 82.18% for the manual analysis.

Table 7.5.: Identified obfuscated names in analyzed obfuscators

Obfuscator	Precision	Recall	F_1	# Of Obfuscated Names
Allatori [All19]	100.00%	100.00%	100.00%	6,821 (98.00%)
DashO [Das19]	100.00%	89.51%	94.46%	17,428 (81.86%)
DexGuard [Dex19c]	95.99%	100.00%	97.95%	8,231 (75.16%)
Jarg [JAR20]	89.93%	100.00%	94.70%	741 (22.74%)
Stringer [str19]	99.90%	99.16%	99.53%	19,293 (78.42%)
yGuard [yGu20]	99.83%	97.45%	98.63%	1,793 (53.09%)
ZKM [Zel19]	99.99%	98.17%	99.07%	19,926 (92.53%)

Table 7.5 shows the precision, recall, and the total unique number of obfuscated names in all obfuscators. While the analysis results of both classification for precision and recall were identical for most analyzed obfuscators, the results for *Allatori* differed between the steps. The anomaly classification identified only 49.95% of the obfuscated names, and only the usage of both classifications identified all of them. However, since we have not identified any libraries in *Allatori*, therefore, we could not test our library mapping.

Observation 15 *In total ObfusSpot identified at least 89.51% of the obfuscated names in all obfuscator codebases with a precision of at least 89.93% (RQ3).*

The main reasons for the false positives and false negatives are short names. The

anomaly classification cannot distinguish between names consisting of three characters, such as `add` or `run` or words accidentally formed by obfuscators. For instance, code that runs code as background tasks often use the word `run` to define the main execution method. However, an obfuscator can also accidentally produce this word by iterating over all possible combinations of three characters.

From the column of obfuscated names, we gained an insight that all obfuscators except *Jarg* and *yGuard* changed most of their names (at least 75.16%). We assume that because both obfuscators are less maintained, they have less obfuscated names than other tools.

Evaluation of Name Obfuscation with Dictionaries While with the previous experiments, we could fully answer *RQ1* and *RQ3*, the answering of *RQ2* with these data sets was not possible because none of them contained names obfuscated using natural language dictionaries. To analyze apps that use such names, we use all 453 apps from the F-Droid data set, which do not contain obfuscated names. We used 302 apps for *LibMapper*'s database and obfuscated the remaining 151 apps. For the obfuscation and analysis, we used *ProGuard* [Pro17] with a name dictionary [Sta20] and executed *ObfusSpot* on the obfuscated apps.

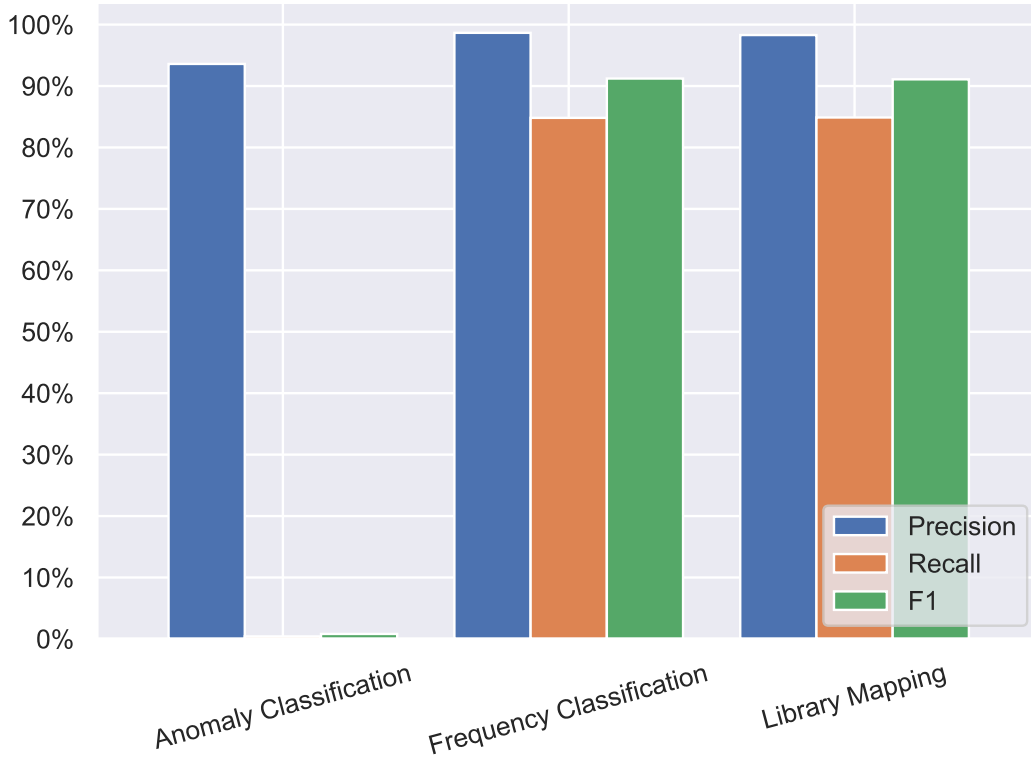


Figure 7.3.: Evaluation of dictionary based obfuscation

Figure 7.3 shows the results for all steps of *ObfusSpot*. All three steps have a high precision of at least 93.6%. However, the recall of the first step amounts to 0.41% because

7. Detection of Obfuscated Names

the step is not suited for names obfuscated using dictionaries and detected only short names with three characters (e.g., `Lee` or `Ann`). Consequently, the F_1 -measure of the anomaly classification is 0.81%.

On the one hand, the precision of the frequency classification is 0.37% higher than the one of the library mapping (98.66% vs. 98.30%); on the other hand, its recall is 0.07% lower than the one of the library mapping (84.82% vs. 84.88%). As a result, the F_1 -measure of the second step is slightly higher than the third step (91.22% vs. 91.09%). However, without library mapping, obfuscated names with low usage frequencies cannot be identified.

Observation 16 *ObfusSpot identifies obfuscated names very accurately and each of its steps is specialized for different name obfuscation areas (RQ2).*

7.3.3. Name Obfuscation in the Wild

In these experiments, we analyze the number of apps obfuscated by developers and the number of libraries available only in obfuscated form. In order to investigate these numbers, we use 100,000 apps randomly selected from AndroidZoo [HAB⁺16].

Apps Name Obfuscated by their Developers To analyze whether developers obfuscated their apps, we filter out all libraries using *AppSeparator* (cf. Section 4.2) and check with *ObfusSpot* whether the remaining code contains obfuscated names.

Observation 17 *Given the results of our tools, we identified 29.33% of apps whose app codebases were obfuscated by developers (RQ4). The names in the main code of 70,668 apps were not obfuscated.*

Previous works identified between 24.92% [WHA⁺18] and 43% [Don18] obfuscated apps in the Google Play store. However, as we showed in Section 3.1, the library-detection approach *LibD* [LWW⁺17] is vulnerable to *Code Restructurings* that is already performed by simple name obfuscation by moving classes into the root package. Consequently, since the approach of Dong et al. [Don18] uses *LibD*, it estimates too high percentages of obfuscated apps (43%).

Additionally, in contrast to our approach, the tool *OBFUSCAN* [WHA⁺18] only considers *ProGuard* [Pro17] as obfuscator. As a result, its identification rate of obfuscated apps is only 24.92%. The slight differences between the results of both tools are explainable since many obfuscators use the same name scheme for their obfuscation. Therefore, both tools produce similar results. However, our approach identified more obfuscated names because it also considers names manipulated by other obfuscators.

Libraries Available only in Obfuscated Form In addition to analyzing apps obfuscated by their developers, we determined which libraries are only available in obfuscated form. This insight is of utmost importance for tools such as *LibMapper* that recovers names of obfuscated code entities.

In order to identify such libraries, we analyzed all packages of the 100,000 apps using *ObfusSpot* and *AppSeparator*. If *AppSeparator* excluded a package from the app code, we considered it as a library package. However, to ensure that developers frequently use this package, we considered only libraries contained in at least four apps.

Observation 18 *For the identification of obfuscated packages, we counted each package as completely obfuscated if ObfusSpot flagged 80% of their names. Given these thresholds, we identified 330 out of 9,203 unique libraries in our data set (3.59%), which are available only in an obfuscated form (RQ5). Additionally, we discovered that 18.53% of the obfuscated names could only be found using our frequency classification combined with LibMapper.*

Table 7.6.: Top 25 libraries available only in obfuscated form

Package Name	Obfuscation Percentage	Unique APKs
org.fmod.*	99.47%	14,292
com.unity3d.player.*	99.59%	14,228
com.milkmangames.extensions.*	82.43%	8,168
com.chartboost.sdk.*	95.16%	6,366
com.facebook.ads.*	98.19%	6,142
com.jirbo.adcolony.*	92.63%	2,931
com.ironsource.mobilcore.*	100%	2,743
com.applovin.impl.*	100%	2,698
com.immersion.hapticmediasdk.*	100%	2,635
com.flurry.sdk.*	100%	2,572
com.vungle.publisher.*	100%	1,581
com.lostpolygon.unity.*	100%	1,482
com.purplebrain.adbuddiz.*	99.01%	1,316
com.andromo.widget.*	100%	896
com.inmobi.*	100%	835
com.moat.analytics.*	94.23%	762
com.makeramen.roundedimageview.*	81.09%	714
com.appnext.ads.*	95.88%	631
com.neatplug.u3d.*	99.81%	516
com.appodeal.ads.*	100%	434
com.yandex.metrca.*	100%	426
com.paypal.android.*	100%	391
com.adcolony.sdk.*	100%	375
com.pollfish.*	97.33%	337
com.facebook.all.*	100%	334

Table 7.6 shows the top 25 libraries that are only available in obfuscated form sorted by the unique number of apps that contain these libraries. The obfuscation percentage indicates how often the package names were only available in obfuscated form. While in the top 25, most of the packages were available only in obfuscated form, some apps contained these packages with less than 80% of names obfuscated. To determine whether these apps contained a non-obfuscated version of the library package, we manually analyzed their code. We identified that in all cases, the analyzed packages contained only

7. Detection of Obfuscated Names

non-obfuscated code provided by the library developers as API for the integration in apps. Obfuscators moved all code that contained the remaining functionality of the library into the root package of the apps. As a result, all of the top 25 packages are only available in their obfuscated form.

From the names of the packages we can deduce that most of these packages contain functionality for ad-networks [App19] such as `com.facebook.ads` and `com.adcolony.sdk`. These networks provide only an API for the app developers and release the remaining code in obfuscated form. As a result, developers cannot understand the libraries' content that they integrate into their apps.

7.4. Threats to Validity

Our studies have one threat in common with all previous works [WHA⁺18, Don18] that is the focus on freely available apps without considering paid ones. While developers of paid apps could have a higher stimulus to obfuscate their apps, in June 2020, only 3.6% of the apps belonged to this category [pai20]. Therefore, we focused only on the larger percentage of the overall apps.

Another threat is the usage of *AppSeparator* to split library code from app code. Since the splitting procedure has no perfect precision, it can lead to app code that is considered as library code or vice versa. However, to the best of our knowledge, *AppSeparator* is the most effective tool for this task.

The second step of *ObfusSpot* could be evaded if an obfuscator distributes all names from a dictionary randomly. Nevertheless, if the distribution is truly random, some of the names occur more than once, indicating a name frequency of obfuscated names.

The 100,000 apps could not be sufficient to generalize the statements made in the evaluation. The Google Play store currently has 2,960,000 apps [num20]. Our data set corresponds to a confidence level of 99% and a confidence interval of about 0.4%. Therefore, the 100,000 apps are sufficient to generalize the statements.

7.5. Conclusion

In this chapter, we presented *ObfusSpot* a name obfuscation detector that uses three different steps to identify various name obfuscation schemes. The first step uses anomaly detection to identify names that do not conform with names in non-obfuscated apps. Most obfuscators use these name obfuscation schemes in their standard configuration to minify names and, therefore, the size of an app. In the second step, *ObfusSpot* identifies obfuscated names using the frequency of the name occurrence that also differ from non-obfuscate apps, mainly if an obfuscator performs overloading. The third step identifies libraries using our tool *LibMapper* (cf. Chapter 6) and generalizes the identified obfuscated names to detect similar naming patterns in the app codebase.

With these three steps, we showed in our evaluation that *ObfusSpot* effectively identifies names obfuscated by multiple free and commercial obfuscators. For this analysis, we downloaded the obfuscators Allatori [All19], DashO [Das19], DexGuard [Dex19c],

Jarg [JAR20], Stringer [str19], yGuard [yGu20], and ZKM [Zel19] and identified at least 89% of the obfuscated names.

Additionally, we analyzed 100,000 apps to identify the number of apps obfuscated by developers and all libraries whose code is only available in obfuscated form. With these results, we close the loop to the name-inference approaches that need to identify new ways to deal with obfuscated libraries.

In the last section, we showed that our 100,000 apps are sufficient to generalize our statements and discussed some of the shortcomings of our approach.

Summary

Analysts develop tools to detect repackaging and malware in Android apps. For the development, they need to analyze different apps, which is hindered by obfuscated names. To support analysts, we developed approaches to identify and recover such names.

In Chapter 6, we presented *LibMapper* to map the original names of libraries on their obfuscated counterparts. For the mapping it uses *AppSeparator* (cf. Section 4.2) to identify all library classes in an app and *LibDetect* (cf. Section 3.2) to identify possible library class candidates for the each of the libraries. Afterward, *LibMapper* explores each class-, field-, and method mapping for each of the suggested class candidates and assigns the one with the highest fitting probability. For the evaluation of *LibMapper*, we compared its precision, recall, and F_1 -measure with the ones of *DeGuard* [BRTV16] the current state-of-the-art name-deobfuscation approach. For the comparison, we downloaded non-obfuscated apps from the F-Droid store [F-D19] and obfuscated them with different name-obfuscation configurations of ProGuard [Pro17]. The results showed that *LibMapper* recovers more obfuscated names than *DeGuard*.

Since many apps contain obfuscated names, it is tedious to find non-obfuscated libraries to supply *LibMapper*'s database with an up-to-date number of libraries. As a consequence, we introduced in Chapter 7 *ObfusSpot* that identifies obfuscated names in three phases. First, it performs an anomaly detection of names that do not conform to the learned patterns of non-obfuscated names. Second, it measures each name's occurrence frequency in an app to identify names that occur more often than a non-obfuscated name. Finally, in the last phase, *ObfusSpot* uses *LibMapper* to identify the naming patterns of obfuscated library names that are already in *LibMapper*'s database to generalize these patterns to unknown library names. For the evaluation of *ObfusSpot*, we downloaded mapping files from GitHub repositories to train and test its classifiers. Furthermore, we tested the generalization of the third phase on the app codebase to identify more obfuscated names than the ones identified by *LibMapper*. All results suggest that *ObfusSpot* is an accurate obfuscated-name detector and identifies more obfuscated apps than comparable approaches. In the end, we analyzed the number of libraries that are only available in obfuscated form by analyzing 100,000 randomly selected apps from the AndroZoo [ABKLT16] data set. Our analysis discovered that mostly ad libraries are only released in obfuscated form.

Part III.

Obfuscated Strings

Obfuscated String Detection and Recovering

String obfuscation is applied by many existing obfuscators [Dex19c, All19, Das19, str19, Zel19]. The presence of obfuscated strings impedes the analysis of apps, e.g., to check their compliance with privacy regulations or to inspect them for detecting malware [RAMB16, MW17]. String obfuscation can hide paths, URLs, and intents that can be used to track the activities of a user or open shells on the user’s device to activate malicious payload remotely.

Opposing prior work [Don18, Mir18, WR17], which stated that strings are often not obfuscated in the wild, in this dissertation, we provide strong empirical evidence (cf. Sections 10.3.2 & 10.3.2) that both malicious and benign apps use it in a wide range. The usage of string obfuscation in benign apps is to a significant extent due to integrated ad libraries – hence, even the app developer may not be aware of its presence. Under these conditions, approaches [ZZPZ19, ZLZ19, NYW⁺18, PWD⁺17, OMA⁺19, FBR⁺16, WL16] that analyze plain strings to identify malware, data leakages, or privacy-related information are ineffective, and techniques for automatically uncovering obfuscated strings are highly needed.

Given that the deobfuscation logic usually is part of the application [WHA⁺18], an analyst can try to debug or run the app with a monkey script. However, such a ”brute-force” testing has serious drawbacks. First, the usage of a monkey script does not guarantee that all execution paths are covered. Second, obfuscated applications could detect the debugging mode activated by monkey scripts and avoid executing the deobfuscation functionality [RAMB16], which is often protected by guards that try to defend against artificial runtime environments [VC14].

Several approaches [Dex20, sim20, Jav20a, Dex19b, RAMB16, BP18, WL18, ZWWJ15] have been proposed to address string obfuscation. But, they suffer from limited scalability and generality. Many of the existing approaches [RAMB16, BP18, WL18] typically alter `if` statements of the target program and run the code with all combinations of values to circumvent defenses and force the execution of all branches. Given that many obfuscators perform automatic string obfuscation on millions of apps, the above approaches are not suited for large-scale analyses. The approach by Zhou et al. [ZWWJ15] slightly reduces the number of executions, but at the cost of generality, as its emulator is fitted to string operations only. In fact, to the best of our knowledge, all works lack a systematic analysis of existing automatic obfuscators and their scope.

To address the above issues, we propose *StringHound*, a novel string deobfuscation technique for Java bytecode. *StringHound* generalizes to different string obfuscations, and to ensure scalability, it executes only the code necessary for the deobfuscation. To

ensure that our approach generalizes over different string obfuscation techniques, we conducted a comprehensive study of such techniques and used the gained insights to guide the design of *StringHound*. In the conducted study, we systematically analyzed strings obfuscation techniques in ad libraries (cf. Chapter 8). These libraries often employ string obfuscation [SGC⁺12, RNVR⁺18, DMY⁺16, SKS16, CFL⁺17] and are, hence, a good source for systematically surveying string obfuscation techniques used in the wild. We ensure that *StringHound* executes only the necessary deobfuscation code by locating the usage of obfuscated strings within the application code.

For locating, we propose two classifiers, one that uses decision trees [Qui86] to identify potentially obfuscated strings, and another one that uses the Spearman correlation [MS04] to identify code of deobfuscation methods. Given that the deobfuscation logic usually is part of the application [WHA⁺18], we propose a specifically targeted slicing technique that includes all program statements which affect the state of an obfuscated string located within a given method. Additionally, *StringHound* extracts the execution context of the deobfuscation logic and injects the slice into it. Through the injection of the slice, countermeasures, potentially introduced by obfuscators, are rendered ineffective. Finally, *StringHound* executes the resulting slice within the extracted context to obtain deobfuscated strings.

In the following, we present our study in Chapter 8, the detection of obfuscated strings in Chapter 9, and the slicing and the recovering of obfuscated strings in Chapter 10. Finally, we summarise all contributions of this part.

8. Study of String Obfuscation Techniques

Many obfuscators [Dex19c, All19, Das19, str19, Zel19] provide techniques that conceal the contents of strings. While many studies [Don18, Mir18, WR17, Jav20a, Dex20, MKS19, PYC⁺19, KNGS18] investigated these techniques, it is unclear whether all variations were explored and whether unknown ones exist in the wild. Without the analysis of string obfuscation in the wild, obfuscated strings containing malicious content may not be detected. Hence, a detailed analysis of string obfuscation is necessary to understand and automatically identify such techniques.

To close this knowledge gap, we systematically analyzed ad libraries since many studies [SGC⁺12, RNVR⁺18, DMY⁺16, SKS16, CFL⁺17] mentioned that these libraries use string obfuscation in various forms and quantities. To identify obfuscated strings, we extracted 640 ad libraries from 100,000 randomly selected apps and analyzed them manually. Obfuscated strings can be represented as constants but also hidden in other data structures. Consequently, to identify obfuscated strings in the manual analysis, we inspect constant strings and methods that process other data structures but return strings. In this analysis, we have identified 21 unique string obfuscations, whose techniques can be categorized into five different concepts for concealing the contents of strings. In the next sections, we describe the used data set in Section 8.1 and our methodology to identify string obfuscation techniques in Section 8.2. Afterward, we give an overview over the identified obfuscation techniques in Section 8.3, and categorize the different concepts of string obfuscation in Section 8.4. Finally, we discuss threats to the validity of our study in Section 8.5, and draw conclusions in Section 8.6.

8.1. Dataset

While some approaches analyzed ad libraries in prior work [LKLT⁺16, LLJG15, NCC14], no data set of all current ad libraries is publicly available. To collect a sample of ad libraries, we analyzed apps that integrate them. As a first step, we collected a list of package names of frequently used ad libraries [LKLT⁺16] and a list of URLs of ad networks [App19]. We reversed the internet domain names (e.g., `youmi.net` \Rightarrow `net.youmi`) of the collected URLs to guess potential package names of ad libraries since it is a common practice to use the reversed URL domain names as package names [ora19]. Next, we collected 100,000 apps by randomly sampling these from the current list of AndroZoo’s database [HAB⁺16]. In these 100,000 apps, we searched for code with the respective package names by comparing them with our collection of package names. To extract the package names of the APKs, we processed each APK with Enjarify [enj17] and extracted the file list from the resulting JAR file (since the JAR file consists of multiple compress files like a ZIP file). Given the file list from a JAR file, we compared the beginning of

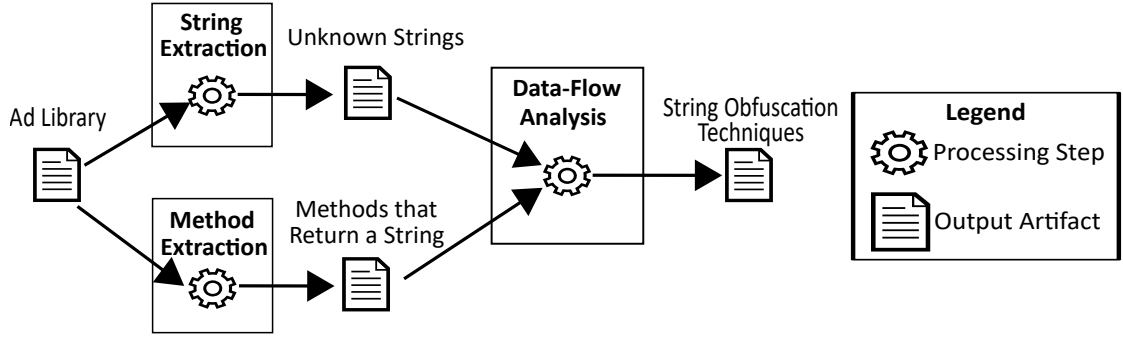


Figure 8.1.: Manual process to identify obfuscated strings

each class file with our collected package list. In this analysis, we identified 640 unique ad libraries distributed across 81,008 individual apps.

8.2. Methodology

For the manual analysis, we performed the processing steps depicted in Figure 8.1. First, we extracted all unknown strings and all method calls that return a string. Second, we checked the data flow to and from the position of a string or a method call. Finally, if the code that is used in the data flow belongs to a string obfuscation technique, we further analyzed its functionality. In the following, we describe the individual processing steps in more detail.

String Extraction For the manual analysis of strings, we extracted all strings from an ad library and manually checked all unknown strings. For instance, the string "`<html>`" were considered as non-obfuscated because it belongs to the starting tag of a HTML document. Additionally, if the string contains multiple words¹ found in a dictionary, it is classified as non-obfuscated. However, we mark strings such as "`i3@5khdsg`" as potentially obfuscated because the usage of this string is not known.

Method Extraction During our manual analysis, we did not focus solely on string constants because, in the obfuscated form, they are often also stored in byte arrays [SKK⁺16]. Hence, we considered different data structures that can be used to hide string representations and further refer to such data structures as obfuscated strings. We analyzed such data structures by using OPAL [EH14] for the identification of all method calls that take non-string parameters as input and return a string. For instance, the constructor of `java.lang.String` takes a character or byte array and returns a string. If the values that flow into such methods are either data structures constructed from constant values or computed using constant values, we marked the data structures as potentially obfuscated strings.

¹Most obfuscators produce strings with unreadable symbols and, therefore, contain no words.

Data-Flow Analysis After identifying unknown strings and method calls that return a string, we investigated the data flow at the positions where these strings and calls were used. If a string directly flows into a method that is not modifiable by the obfuscator (e.g., `System.out.println`), it is considered as non-obfuscated. For all other marked strings and method calls, we performed a manual data flow analysis to check whether these represent real obfuscated strings. If a marked string or method call is not processed by deobfuscation logic, we ignored such cases in further manual analyses.

If an obfuscated string was identified, we analyzed the surrounding code to determine the used technique. This code is placed into apps to deobfuscate the string during the app execution [WHA⁺18]. For the analysis, we used CFR [CFR19] to decompile the code of a JAR file and load the resulting Java files into IntelliJ IDE [int20b] to navigate through the classes and their dependencies.

Table 8.1.: String obfuscation techniques found in ad libraries

Example Package	Cipher	Encoding	Countermeasures	Count
a.a.a	AES	-	Serialized Object	3 (1.27%)
br.com.tempest	Bit	-	Key Changed by Switch Statements	3 (1.27%)
br.com.tempest	Bit	-	Key is the Signature of Stack Calls	3 (1.27%)
br.com.tempest	Bit	-	Stack Calls	3 (1.27%)
br.com.tempest	Bit	-	Switch Statements	3 (1.27%)
com.intentsoftware	-	Base85	-	3 (1.27%)
com.youmi	-	BigInt33	-	4 (1.69%)
com.adcolony	-	URLEncoder	-	5 (2.12%)
com.apptracker	Bit	-	Two Keys	8 (3.39%)
com.champspire	-	Base64	-	8 (3.39%)
com.google.android	Bit	-	Stream Transfer	8 (3.39%)
com.mnt	Bit	Base64	Key is the Index of Byte Arrays	8 (3.39%)
com.tnkfactory	Bit	-	Object Initializer	8 (3.39%)
com.adlib	Bit	-	Two Methods	12 (5.08%)
cn.pro.sdk	Bit	-	Byte Arrays	13 (5.51%)
com.ironsource	-	Split	-	16 (6.78%)
com.google.android	AES & Bit	Base64	Static Initializer	22 (9.32%)
com.applovin	Bit	-	Key Hidden in Byte Arrays	24 (10.17%)
com.mt.airad	DESede	Base64	-	24 (10.17%)
com.vpon.adon	DESede	-	-	28 (11.86%)
com.waystorm.ads	Bit	Base64	Key Management Calls	30 (12.71%)

8.3. Overview of Identified Techniques

The analysis of ad libraries has shown that only about 37% (236 of 640) of these libraries contain obfuscated strings. From these obfuscated strings, we identified 21 unique string obfuscation techniques and present them in Table 8.1. While some identified techniques are custom-made, some of them are used by the state-of-the-art obfuscation tool manufacturers such as, DexGuard 5.5.41 [Dex19c], Allatori 6.8 [All19], DashO 9.2 [Das19], Stringer 3.0.5 [str19], ZKM 12.0 [Zel19], and Shield4J [Shi20]. For each technique, we

8. Study of String Obfuscation Techniques

show whether it uses a cryptographic cipher, an encoding, countermeasures, or a combination of the former to evade the detection by static/dynamic analyses. Additionally, we list the count (Count) of each string obfuscation technique across all 236 ad libraries.

Cryptographic Ciphers are used by various obfuscation techniques, ranging from simple self-made to standardized cryptographic algorithms that encrypt strings. Many of the identified custom-made algorithms convert the ASCII characters of a string into bytes, apply to these bytes logical bit operations (Bit) with a fixed value as the key of the algorithm, and convert the resulting bytes back into a concatenated string. Most such bit operations are either XOR operations or a combination of other bit operations, resulting in an XOR operation. For revealing the original string, the integrated deobfuscation logic often uses the same values with the same operations as for the encryption.

Most of the observed standardized cryptographic algorithms originate from the Java Cryptography Architecture (JCA). During our study, we found the algorithms DESede and AES [Cry20]. They either were called directly from the implementation of the JCA, or the implementation of the algorithms was extracted and integrated into their app code, as described for *Fake Types* in Section 2.2. For revealing the original string, the integrated deobfuscation logic uses the decryption algorithm with the included key.

Encodings similar to cryptographic ciphers are used by obfuscators in different forms, and they also vary between simple custom-made techniques and standardized algorithms that encode strings. For example, an observed custom-made technique splits (*Split*) a string into its characters and puts the characters together in an arbitrary order. The technique attaches an additional flag to each character, which is used to restore the original order. Another technique converts the characters of a string into bytes, transforms the bytes from base 10 to base 33, and converts the newly recovered bytes back into a string. For restoring the original string, the technique uses the Java class `BigInteger` (*BitInt33*) to interpret numbers directly with the specified base.

The most commonly used standard encoding algorithm is Base64 [Bas19a]. The strings that are obfuscated with this algorithm are easily recognizable because most of them end with an equal sign. This sign is used to pad the string if it is too short for the encoding. For example, the string "Hello" is changed to "SGVsbG8=". The Android class library contains an implementation of the Base64 algorithm, and the analyzed obfuscators often call this implementation. A less frequently used algorithm is URL encoding, which is implemented in the class `URLEncoder` [URL20] of the Android class library. As with ciphers, obfuscators either call the encoding algorithms directly or extract the implementations of the algorithms and integrate them into their code, as described for *Fake Types* in Section 2.2. Finally, we identified the implementation of a Base85 algorithm [Bas19b], which can be deobfuscated, similarly to the Base64 algorithm.

Countermeasures are used by obfuscators to prevent the extraction of the original string or to prevent the unauthorized execution of the deobfuscation logic. In the fol-

lowing, we briefly describe all identified countermeasures and which deobfuscators they might evade:

Serialized Object: During our analysis, we found one technique that loads at runtime a serialized class object that contains the deobfuscation method. Subsequently, the deobfuscation method has to be called through reflection to reveal the content of a string. This technique evades current deobfuscators (*Dex-Oracle* [Dex20], *JMD* [Jav20a]) that rely exclusively on the identification and execution of deobfuscation methods.

Switch Statements: Two obfuscation techniques perform the deobfuscation of strings before the class containing the obfuscated strings is used. These obfuscated strings are stored in an array of strings. This array is accessed during the initialization of the class by a method to deobfuscate the strings in a switch statement according to a predefined order. While one technique always uses the same key to deobfuscate all strings, the other one changes the key in each case of the switch statement. Afterward, both techniques store the resulting strings back into the string array, and each method that uses a deobfuscated string needs to access this array. These techniques evade deobfuscators (*Dex-Oracle*, *JMD*) that search for an explicit deobfuscation method.

Stack Calls: Two obfuscation techniques include the calling context (e.g., method name and class name) of the deobfuscation method into their logic. While one technique checks the calling context in a conditional statement, the second one uses the calling-context information as part of the deobfuscation key. Both techniques evade deobfuscators (*Harvester* [RAMB16], *Dex-Oracle*, *JMD*, approach by Zhou et al. [ZWWJ15]) that execute the deobfuscation logic without a specific context. However, only one of the obfuscation techniques enforces the extraction of the context for slicing approaches because the context is a part of the deobfuscation.

Two Keys: One obfuscator uses two different keys for the string deobfuscation. This usage of keys evades deobfuscators (*JMD*) that either try brute force guessing the key or extract only one key to uncover obfuscated strings.

Stream Transfer: One obfuscation technique uses covert channels to transfer obfuscated strings to their deobfuscation logic. This technique transfers an obfuscated string using output streams. Therefore, it evades deobfuscators that track only constant strings without analyzing data flows to streams, such as the approach of Zhou et al. [ZWWJ15].

Byte Arrays: Two of the analyzed obfuscation techniques use byte arrays as data structures to hide obfuscated strings and, thus, evade tools [Don18, Mir18] that do not analyze such structures. We found the technique by analyzing data structures that are converted to strings.

Object Initializer: One obfuscation technique inserts, into the root package, a specific class that declares only a constructor and one additional method. The constructor

8. Study of String Obfuscation Techniques

initializes a key, and the additional method uses this key to deobfuscate all strings that were obfuscated using this object. This technique evades deobfuscators (*Dex-Oracle*, *JMD*) that execute only static methods.

Two Methods: One obfuscation technique uses two different methods to deobfuscate a string. While the first method takes the obfuscated string as a parameter and returns an intermediate character array, the second one takes this intermediate character array and returns the deobfuscated string. This usage evades deobfuscators (*Dex-Oracle*, *JMD*) that execute only one deobfuscation method to uncover obfuscated strings.

Static Initializer: We discovered a byte array that is used in the method to initialize classes (static initializer). This byte array contains a key that is used to deobfuscate strings. This practice evades tools (*JMD* [Jav20a]) that extract only the logic of a deobfuscation method without considering the static initializer.

Key Management Calls: One obfuscation technique initializes the key used for deobfuscation directly before calling the deobfuscation logic. These keys are stored in fields of the same class object that uses the deobfuscated string and calls the deobfuscation method. This technique hinders deobfuscators (*Dex-Oracle*, *JMD*) that do not handle such initialization.

As depicted in Table 8.1, different combinations of cryptographic ciphers, encodings, and countermeasures are used as techniques for string obfuscation. We refer to these combinations as obfuscation schemes. Some of these techniques are used in state-of-the-art commercial obfuscation tools, and developers most commonly use them to obfuscate strings in Android and Java apps.

8.4. Identified Concepts of String Obfuscation

Given the 21 identified string obfuscation schemes, we determined five concepts used to obfuscate strings in the wild. In the following, we elaborate on each of these concepts and discuss whether current deobfuscators could evade these concepts.

Hide Obfuscated Strings: While some works [Don18, Mir18, WR17] identify obfuscated strings, only a few works [Dex20, Jav20a] can identify data structures that hide the presence of obfuscated strings. These works try to identify deobfuscation methods that transform the byte or character arrays and return the original content as a string. Since these works focus solely on data structures that are used by known obfuscators, they miss data structures that are used in the wild, such as strings hidden in streams.

Hide Deobfuscation Logic: Hiding the deobfuscation logic, e.g., through serialized objects, can circumvent the detection of obfuscated strings. However, since those obfuscation schemes use string constants to represent the obfuscated strings, the

hidden code can be identified and extracted. While the usage of the string representation can be avoided by hiding the obfuscated string in other data structures, the procedure that hides the deobfuscation logic itself can be an indicator of the usage of obfuscated strings. For instance, since the usage of serialized objects is not common in Android apps, the code to deserialize the deobfuscation logic can be used to identify the obfuscation scheme.

Hide Data Flow to Deobfuscation Logic: Some obfuscation schemes hide the data or code of the deobfuscation (e.g., key, obfuscated string, or parts of the deobfuscation logic) in other fields or methods. For instance, one scheme hides an obfuscated string by using output streams. Since the transfer of obfuscated strings is performed in the same method as the rest of the deobfuscation logic, the extraction of this logic is straightforward. However, the extraction of deobfuscation data is only possible because it is kept in the same class as the usage of the deobfuscated string. We assume that this shortcoming results from the usage of fully-automated obfuscators [Dex19c, All19, Das19, str19, Zel19, Shi20].

An obfuscator has to integrate the deobfuscation logic of a string into the application [WHA⁺18]. This logic could be extended with new parameters, fields, or methods to increase its complexity. This extension would lift the deobfuscation logic to an interprocedural problem because new methods need to be analyzed, and parameters and fields are initialized in other methods.

The insertion of **new parameters** to a method requires that the obfuscator identifies all calls of this method and adapts all calling methods (e.g., stack layout) to add the parameter and its corresponding argument. However, to find all method calls, the analysis of an obfuscator has to deal with polymorphism, intents, and reflection. Reflection is frequently used [RKE⁺19] in the wild and not completely resolved by current analysis. As a result of the above requirements, an automatic obfuscator cannot lift the logic to an interprocedural problem without unambiguously identifying all method calls.

Inserting **new fields** is more convenient than inserting new parameters. However, the obfuscator must initialize the new fields before the deobfuscation logic can access it. Initializing the new fields directly before each call of the deobfuscation method would lead to the same problem as with new parameters (i.e., identification of all calls). Thus, studying the JVM specification, we identified only two other options to execute the field initialization before the deobfuscation method. In the first option, the JVM executes the static initializer before the code of any other method. Thus, an obfuscator can initialize a field in the static initializer. The second option is the execution of the constructor/object initializer before any other instance method. Thus, if the entire logic is in instance methods, an obfuscator can use a constructor for the field initialization. However, both possibilities reduced the analysis space to the class code that contains the deobfuscation logic.

An obfuscator can insert **new methods** at three different positions to increase the deobfuscation complexity. The first position hides an obfuscated string in the

8. Study of String Obfuscation Techniques

last called-method of newly inserted methods that transitively call each other in a chain. However, if a deobfuscator identifies the first method call of the chain, then it can be executed to get the obfuscated string. Second, the deobfuscation logic can be hidden in a chain of calls, as performed by the *Two Methods*-technique. However, if a deobfuscator identifies the sink of the deobfuscated strings, then it can call the method that contains the sink to retrieve this string. Finally, the sink of the deobfuscated string can be hidden in a chain of calls. However, not every sink is suitable to be embedded in such a chain of calls because techniques like *Stack Calls* use the information of the surrounding method.

The discussion above indicates that interprocedural obfuscation is hard for automatic obfuscators. It can only be performed in a limited environment or requires input from the obfuscation user. For instance, to lift the deobfuscation logic to an interprocedural problem, a user could add a parameter to a method by identifying all calling methods and adding an argument to the method calls.

Protect Deobfuscation from Execution: Some obfuscation schemes protect either the whole or a part of the deobfuscation logic so that it can only be executed if certain conditions are met. While the usage of `if`-branches is circumvented by current works such as Harvester [RAMB16], the usage of the surrounding environment as deobfuscation keys, such as performed by *Key is the Signature of Stack Calls*, are not handled. However, since the obfuscation scheme cannot control the entire stack trace of the currently executed method, these countermeasures can be circumvented by keeping all information unchanged that is accessible via stack traces.

Hide Deobfuscation Keys: While some schemes use an openly accessible deobfuscation key, others either hide the key in other data structures or compute it during runtime. However, a deobfuscator can easily extract the location of the key because most schemes keep them close to the deobfuscation logic. Some schemes separate the computation of the deobfuscation key from the main logic to increase the complexity of the entire process. However, if an obfuscation scheme moves the computation of a key to extend the logic to an interprocedural problem, it has to tackle the limitations described in *Hide Data Flow to Deobfuscation Logic*. These limitations may be responsible that the analyzed obfuscation schemes do not move the deobfuscation key beyond the borders of the class that use the obfuscation string. Given the scope of the entire deobfuscation logic, it can be extracted by analyzing the class that uses the deobfuscated string.

Observation 19 *None of the identified techniques requires a broader focus than the class that uses the deobfuscated string. All analyzed obfuscation schemes initialize the deobfuscation logic within the same class that contains the logic. Thus, no heavyweight interprocedural analysis seems necessary for our data set.*

8.5. Threats to Validity

In this section, we discuss the two major threats to the validity of our findings and our methodology. Ad libraries may not contain all string obfuscation techniques or variations of them. To mitigate the threat that ad libraries do not contain all variations of string obfuscation, we chose ad libraries from different domains, in different sizes, and with different complexities. Furthermore, we used obfuscation schemes that are applied by current obfuscators. Developers who want to hide their strings will most likely use current obfuscators instead of a custom made scheme.

During the manual analysis of the extracted strings, we might have missed obfuscated strings because our indicators may have been based on subjective factors. To ensure that we did not miss obfuscated strings, we checked the data flow of each suspicious string. Additionally, we reduced the threat by picking 200 randomly extracted strings and classifying each of them with two authors of the *StringHound* publication [GMB⁺20]. The analysts classified the strings separately without the knowledge of the mutual results. The comparison of the results showed a 100% match.

8.6. Conclusion

In this chapter, we identified 236 unique ad libraries that we have extracted from 100,000 randomly selected apps from AndroZoo [HAB⁺16] to examine techniques that obfuscate strings. According to previous works [SGC⁺12, RNVR⁺18, DMY⁺16, SKS16, CFL⁺17], many of the ad libraries contain obfuscated strings, and therefore, we used ad libraries for the identification of string obfuscation techniques. To identify obfuscated strings, we scanned all constant strings for abnormal usage of non-alpha-numeric characters. Additionally, we analyzed all code locations that get non-string arguments but return strings to identify deobfuscation code.

Using this procedure, we discovered 21 unique string obfuscation techniques and debugged their functionalities. In particular, we have focused on mechanisms used for the protection of obfuscated string from unauthorized deobfuscation. The study showed that all identified string obfuscation techniques place their deobfuscation logic in the same class as the obfuscated string, making its logic easy to identify.

In the end, we discussed different mechanisms that protect obfuscated strings and the shortcomings of their current implementations against automatic analyses. Furthermore, we argued that fully-automated string obfuscation is hard without the interaction with the obfuscation user.

9. Detection of Obfuscated Strings

Previous studies [Don18, Mir18] showed that many apps contain obfuscated strings. While these studies identify whether an app uses obfuscated strings, their approaches cannot pinpoint each string or the code that is used to deobfuscate the string [WHA⁺18]. These approaches have two main limitations that hinder the pinpointing of obfuscated strings. The first limitation is the high aggregation level of the features used to analyze apps. For instance, the approaches extract all strings of an app and calculate the distribution of each character over all strings instead of calculating information for each individual string. The second limitation is that most approaches solely focus on string representations in apps without considering other data structures, such as byte arrays. As we identified in Chapter 8, at least two obfuscation schemes use other data structures.

Currently, most approaches cannot tackle the first limitation. However, some approaches [Dex20, Jav20a] partially mitigate the second limitation by identifying deobfuscation methods that process other data structures. These approaches use the method signatures used by known obfuscators. However, without considering the method body, therefore, the approaches cannot handle inlined deobfuscation logic used by more advanced obfuscation schemes. To avoid both of the limitations, we introduce two classifiers that complement each other. The first classifier (*String Classifier*) can pinpoint an individual obfuscated string by calculating features for each string. The second one (*Method Classifier*) identifies the usage of other data structures by scanning each method for instructions, which are used in deobfuscation logic. The *String Classifier* uses decision trees to dissect the characteristics of obfuscated strings, and the *Method Classifier* matches instructions of known deobfuscation logic using the Spearman correlation.

We evaluated both classifiers by downloading the newest apps from the F-Droid store, obfuscating their strings, and executing both classifiers on the strings and deobfuscation logic. The results show that both classifiers identify their targets with high accuracy.

The following sections introduce related work for both classifiers in Section 9.1, our *String Classifier* approach in Section 9.2, and its evaluation in Section 9.3. Furthermore, Section 9.4 presents the *Method Classifier*, followed by its evaluation in Section 9.5. Finally, Section 9.6 discusses threats to the validity of our experiments, and Section 9.7 draws a conclusion for this chapter.

9.1. State-of-the-Art Detectors of String Obfuscation

In this section, we describe related approaches for identifying obfuscated strings and deobfuscation methods [Don18, Mir18, WR17, Jav20a, Dex20].

9. Detection of Obfuscated Strings

Dong et al. [Don18] identify apps with obfuscated strings by counting the occurrence of 3-grams in each string and using these counts to train a Support Vector Machine (SVM) classifier [Wan05]. The SVM is then used to distinguish between apps with obfuscated strings and apps without obfuscated strings. While this approach identifies whether an app contains obfuscated strings, it cannot pinpoint an individual obfuscated string because it operates on all strings of an obfuscated app.

Mohammadinooshan et al. [MKS19] identify obfuscated strings by training an n-gram model on non-obfuscated text from different languages and checking whether the strings of an app can be associated with a specific language. If the string does not belong to one of these languages, it is marked as obfuscated.

While this technique can pinpoint obfuscated strings, it does not consider obfuscated strings hidden in other data structures. Additionally, some strings in apps that are non-obfuscated can have a different appearance than texts in an article. Therefore, the approach could confuse these strings with obfuscated ones. For instance, strings that contain XML structures are rare in articles, and because of their extensive usage of special characters, they can be confused with obfuscated strings.

AndrODet [Mir18] identifies apps with obfuscated strings using seven classifiers. These classifiers are trained with multiple aggregated features, and their classifications are combined using limited random search [Pri77]. Each newly analyzed app is used by *AndrODet* to improve its models. However, *AndrODet* only identifies if entire apps are obfuscated and cannot pinpoint the exact strings because it aggregates its features using all strings in an app.

Wang et al. [WR17] identify apps with obfuscated strings by determining the used obfuscator and even the used configuration. For this purpose, the approach extracts all names and strings, filters them, counts their occurrence, and uses the occurrences to train a SVM for each configuration of known obfuscators. The resulting SVMs can identify if a configuration enables string obfuscation to manipulate an app. Unlike *String Classifier*, the described tool only identifies if an entire app contains obfuscated strings but cannot pinpoint which of the strings are obfuscated.

Park et al. [PYC⁺19] identify apps with obfuscated strings by decompiling all Java classes from an app, treating the classes as text documents, and classifying the documents using various machine learning techniques. The approach can identify different obfuscation techniques applied to all classes of an app, such as renaming, string obfuscation, control-flow obfuscation, and the usage of reflection. However, it cannot pinpoint the exact location of an obfuscated string because it uses all classes to identify the various obfuscation techniques.

Kaur et al. [KNGS18] identify obfuscated strings by transforming the bytes of an app's codebase into an image and using image classification to determine the used obfuscation

technique. As the approaches of Park et al. [PYC⁺19] and Wang et al. [WR17], this approach can identify the different obfuscation techniques used to manipulate an app. However, this approach cannot pinpoint the exact location of an obfuscated string.

JMD’s Deobfuscation Method Detector [Jav20a] identifies deobfuscation methods of Zelix KlassMaster (*ZKM*) [Zel19], *DashO* [Das19], *Allatori* [All19], and two generic methods. The detector identifies them using the signatures of the methods, and the direct predecessor instructions of their method call.

While the matching of method signatures can identify most of the methods, it fails to identify inlined or hidden logic. Some obfuscators hide the appearance of the deobfuscation method or integrate the deobfuscation logic into the method that used the original string. Additionally, an obfuscator can evade the check of the direct predecessor instructions of calls by using more parameters than identified by the detector. Our *Method Classifier* does not have such limitations because it focuses only on the instructions of the deobfuscation logic, not the entire method or its signature.

Dex-Oracle’s Deobfuscation Method Detector [Dex20] identifies the deobfuscation methods of *DexGuard* [Dex19c] and a generic method based on method signatures.

In contrast to our *Method Classifier*, this detector accepts no variations of the parameter or return types. For instance, it identifies only the deobfuscation method of *DexGuard* that takes three `int` parameters and returns a `java.lang.String` and cannot handle if *DexGuard* inlines the deobfuscation logic. The more generic approach of *Dex-Oracle*’s detector is also restrictive because it considers only method signatures that take a single parameter and return a `java.lang.String`. In contrast, our *Method classifier* identifies all kinds of deobfuscation logic, even the inlined logic.

Discussion To recap, all approaches that identify obfuscated strings have significant limitations. They can only detect if an app contains obfuscated strings but fail to pinpoint the obfuscated strings. Additionally, most approaches do not handle obfuscated strings hidden in other data structures.

The detectors that try to identify deobfuscation methods by their signatures are restricted if these signatures vary. Additionally, these detectors cannot handle inlined deobfuscation logic.

9.2. String Classifier

The *String Classifier* extracts different features from a string to train a model that separates previously unseen strings into obfuscated and cleartext classes. Figure 9.1 shows the process to classify all strings of a given APK. Before the *String Classifier* can classify strings, it needs to extract all strings and their containing classes. We extract these classes to check if they use cryptographic libraries and process the strings to extract a feature vector for each string. Given a data set of obfuscated and non-

9. Detection of Obfuscated Strings

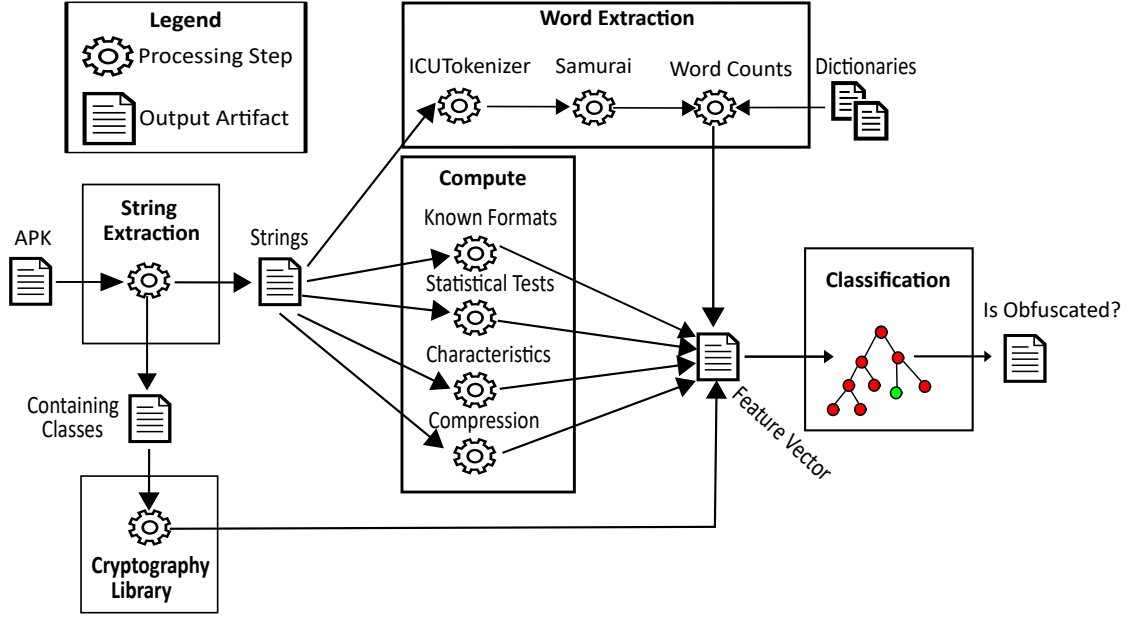


Figure 9.1.: Process of *String Classifier* to learn and classify obfuscated strings

obfuscated strings, we calculate the feature vector for each string to train a classifier. Afterward, the classifier decides for each unseen string if it is obfuscated or cleartext.

For the classification of individual character strings, we selected our features based on the observations in our study (cf. Chapter 8) and basic knowledge of cryptanalysis [Kah96, cry19, LH07]. Using the combined knowledge, we extracted 49 different features from a collection of strings. Using these features, we evaluate, in Section 9.3.1, multiple classifiers to find the most suitable one for our approach.

We extracted the following features to train the selected classifiers:

Formats: In the study presented in Chapter 8, we observed that obfuscated strings contain non-alphanumeric characters. Nevertheless, we cannot classify a string as obfuscated just because it contains non-alphanumeric characters – plain strings of certain formats may also use such characters. To avoid matching such plain strings, we use 27 different regular expression patterns to flag format usages such as XML (e.g. `</th>`) and HTML colors (e.g. `#FFAE40`) in the feature vector. These flags give the model a hint that the analyzed string might not belong in the obfuscated class. However, these hints should not be confused with filtering, as they are only a part of the classifier’s decision.

Statistical Tests: Previous statistical analyses of encryption mechanisms [Kah96, cry19, LH07] show that obfuscated strings often have a random (close to equal) distribution of characters. We use random distribution as a discriminating feature to distinguish between obfuscated and other strings with special characters. To check whether the distribution of the characters is random, we use three different

measures because we encountered that each one is suited for different scenarios. For instance, previous works [LH07] used the *Normalized entropy* to identify encrypted malware. We reuse it to identify encrypted strings. While the *Chi-squared* test measures the deviation of the characters from the equal distribution, the *average distribution* measures whether an obfuscator rotated the characters of a given string (e.g. caesar cipher [Kah96]) or it belongs to a language.

Compression Rate: If our classifier uses only the statistical tests, it may confuse obfuscated strings with compressed data, such as images compressed using JPEG and stored in strings. To correctly identify such strings, we compressed them and compared the length of the compressed string against the original string length. Based on cryptanalysis knowledge [cry19], the comparison will show that both lengths do not differ if the original string already consisted of compressed data.

Cryptographic Libraries: Even if a string contains seemingly obfuscated content, it could have a legitimate use case. For instance, cryptographic libraries use byte-encoded strings to initialize their algorithms, and this may cause false positives because they are similar to obfuscated strings. To avoid matching such encoded strings, we check if the code that contains such a string belongs to a cryptographic library class. For instance, classes that implement the Java Cryptography Architecture (JCA) could contain strings which initialize cryptographic algorithms.

Word Counts: The study in Chapter 8 revealed that obfuscated strings contain few or no words. As a result, we use dictionaries to check whether a string contains words. To perform this check, we need to split the string into text blocks. However, some languages (e.g. Chinese) do not use separators (e.g. white spaces). Furthermore, some strings contain identifiers such as `getLength`. For the parsing of such strings, our process uses an *ICUTokenizer* [ICU19] to split text blocks either by spaces or for languages that do not use spaces by heuristics. If a string contains identifiers, our process splits these identifiers using *Samurai* [EHPVS09]. It splits identifiers by different character cases and frequently used words. Afterward, we use all split blocks to check in multiple dictionaries if these blocks match real words and calculate the word features based on these checks.

String Characteristics: We use five features from *AndroDet* [Mir18] that identifies if an app uses string obfuscation. However, *AndroDet* aggregates all string features over the app and is therefore not able to classify individual strings. Additionally, we combine the features of *AndroDet* with eight additional ones that calculate different character distributions, e.g., character counts, digits.

Table 9.1 shows all features that are used in *String Classifier* to decide whether a string contains obfuscated content. Given the descriptions of all features and their extraction procedure, we describe in the next section the data set used for training and evaluation of our *String Classifier*.

9. Detection of Obfuscated Strings

Table 9.1.: Feature list for the detection of obfuscated strings

Category	Name	Description
Formats	27 Known Formats	User Agents, URLs, Character set of regular expressions, Network protocols (e.g., WiFi), Common OS commands, JSON format, Encodings (e.g., UTF-8), E-Mail format, DTD, HTML Colors, Class Path Format, SQL Queries, Keywords for seven programming languages, country names, XML format, IP format, HTTP state format, Multiple Date formats, Numeric formats, Cryptographic primitives, Mobile phone brands, HTML special characters (e.g., uuml), String-encoded certificate format, String-encoded Android certificate format, Private/Public key format, String signatures of social network apps, String-encoded images (e.g. JPEG),
Statistical Tests	Chi-squared Test	Tests if all chars in the given string are equally, distributed indicating a random distribution.
	Average Distribution	The average distribution which is close to the Gaussian distribution for plain strings,
	Normalized Entropy	The normalized entropy of the strings.
Compression Rate	GZIP	The rate of the GZIP compression,
Cryptography Library	JCA	The string is used in a known crypto library.
Word Counts	Dictionaries of the 54 Languages [Mil95, dic20]	The shortest word length, the largest word length, the number of words, the number of unique words from a multiple language dictionary.
String Characteristics	AndroDet[Mir18]	Number of equals, Number of dashes, Number of slashes, Number of pluses, Sum of repetitive characters.
	Character Counts	Number of vocals, Number of consonants, Number of digits, Number of characters, Number of unique characters, Number of non letters, Maximum number of consecutive characters, Maximum occurrences of the same character

9.3. Evaluation

For the evaluation of *String Classifier*, we investigated the following research questions:

RQ1: What is the most suitable classifier for our data?

RQ2: How effective is our model on unknown strings?

RQ3: What strings are incorrectly classified?

To answer these questions, we conducted three experiments. In the first experiment, we trained eight classifiers whose capabilities enable the processing of our data. The second experiment calculates the precision, recall, and F_1 -measure on our data using the chosen classifier from the first experiment. Lastly, we use regular expressions to investigate the false positives and false negatives of the second experiment.

We performed our experiments using a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The experiments were executed on the server using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory for Weka 3.9.4.

For the training and evaluation of all used classifiers, we downloaded the newest versions of all 1,879 apps from F-Droid [F-D19]. We chose the F-Droid store because it consists only of open-source software. While this data set should be free from string obfuscation, we also ensured its absence by checking all strings following the technique in Chapter 8. Given these apps, we use them as ground truth of plain strings. The absence of other string obfuscation allows us to perform our experiments without dealing with the influence of previously existing string obfuscation artifacts. While the standard configuration of the Android build process might obfuscate names due to the integration of *ProGuard*, it does not use string obfuscation [Pro20b].

To extract our final data set, we generated 1,918,687 obfuscated and the same amount of non-obfuscated strings. We extracted the non-obfuscated strings from all downloaded F-Droid apps and obfuscated the apps using the 21 obfuscation schemes identified in our study (cf. Chapter 8) to get the obfuscated strings. The usage of the schemes yielded 32,379 obfuscated apps¹ from which we extracted all strings. As a result of the obfuscation process, we acquired significantly more obfuscated strings than cleartext strings. To avoid a bias towards obfuscated strings, we took all strings from the non-obfuscated apps and randomly selected the same number of strings from the obfuscated ones. For our experiments, we used 80% of these strings for training and testing and 20% for validation.

9.3.1. Classifier Comparison

For the comparison of the most suitable classifier, we analyzed Naive Bayes, Multilayer Perceptron, Logistic Regression, Random Tree, Random Forest, and REP Tree from

¹We could not obfuscate every app with each obfuscation scheme due to version incompatibilities between acquired tools and APKs.

9. Detection of Obfuscated Strings

Weka's [WF02] collection. While we trained most classifiers using their default values, we evaluated the Multilayer Perceptron using three different epoch values because of their impact on the classifier's performance.

We trained and tested the eight classifiers using 10-fold cross-validation [HTF09] on the training set of strings. Given the statistics of the cross-validation, we compare the correctly-classified instances, and their root mean squared error of the classifiers. The root mean squared error is a measure for the false classifications of a model.

Table 9.2.: Selection of classifiers for obfuscated strings

Classifier	Correctly Classified	Root Mean Squared Error
Naive Bayes	79.17%	42.1%
Multilayer Perceptron (one epoch)	84.43%	28.16%
Multilayer Perceptron (five epochs)	91.13%	20.67%
Multilayer Perceptron (10 epochs)	85.09%	26.89%
Logistic Regression	95.04%	19.68%
Random Tree	98.18%	12.67%
Random Forest	98.35%	11.56%
REP Tree	98.37%	11.51%

Table 9.2 shows the correctly classified and falsely classified measures of each of the eight classifiers. While the Multilayer Perceptron seems to improve after five epochs, the classification performance with more epochs decreases with each epoch until it reaches a similar value as with one epoch. Both the Random Tree and Random Forest achieve more than 98% correctly classified instances.

Observation 20 *Using the REP Tree yields the best score with our training data. The correctly classified instanced of REP Tree amounts to 98.37% and the false classification rate (root mean squared error) to 11.51%. The results in Table 9.2 show that REP Tree is the most suitable classifier to process our data (**RQ1**), and therefore, we select it to classify if strings belong to the obfuscated or cleartext class.*

9.3.2. Effectiveness of the String Classifier

Given that in the previous section, the REP Tree model yielded the best results, we evaluate it on the validation data set to measure its performance on new strings.

Table 9.3.: Results on the validation set

Measure	Percentage
Precision	98.79%
Recall	89.75%
F_1	94.05%

Observation 21 *The validation results in Table 9.3 reveal that the REP Tree model has a precision of 98.79%, a recall of 89.75%, and a F_1 -measure of 94.05%. These results show that our model is very effective (RQ2).*

While our REP Tree model is very powerful in classifying unknown strings, it still has false classifications that we examine in the following section.

9.3.3. Falsely Classified Strings

The results of the validation set contain 4,600 false positives and 26,780 false negatives, which we show in Table 9.4 grouped by regular expressions. While we group the false negatives (FN) by regular expressions that identify the encodings used by the obfuscation schemes, we group the false positives (FP) by patterns that categorize their data.

Table 9.4.: False-classified strings grouped by regular expressions

Regular Expression	Percentage	Category
Base64	3.33	FN
URLEncoder	4.73	FN
BigInt33	14.18	FN
Short Strings	77.76	FN
JSON Format	0.02	FP
Numbers	0.09	FP
IP	0.13	FP
Country Sings	0.24	FP
Encodings	0.48	FP
Paths	0.61	FP
Hash Sums	2.2	FP
HTML	2.63	FP
Cryptographic Algorithms	4.2	FP
Commands	5.63	FP
Brands	7.7	FP
Date Formats	11.63	FP
Regular Expressions	13.96	FP
Mix of Languages	50.48	FP

During our analysis, we identified two root causes for false negatives. The first cause is that our approach cannot distinguish between legitimate usage of hash sums in cleartext or the usage of hash sums in obfuscated strings such as produced by the *BigInt33* encoding. The same issue we observed for false positives of hash sums. For instance, a human reader cannot distinguish if a hash sum, such as AB123FF10C, belongs to an obfuscated string or not.

The second, more prevalent cause results from the obfuscation of short strings. These obfuscated strings may contain digits, valid words, or characters, which is similar to short cleartext strings. As a result, *String Classifier* cannot distinguish short cleartext from short obfuscated strings. For instance, an obfuscated string may accidentally contain valid words because some obfuscation schemes use bit operations that might change the

9. Detection of Obfuscated Strings

value of a character so that the new character lies in the code point section of Chinese characters. Since some Chinese characters represent entire words, the classifier cannot distinguish between obfuscated and cleartext strings.

As the leading causes of false positives, we identified strings containing regular expressions, date formats, and known abbreviations such as `WIFI` mixed with foreign languages (e.g. Chinese). Since our dictionaries do not contain abbreviations and not all foreign languages, their appearance causes 50.48% of the false positives. In addition to the previously-mentioned categories, short strings such as HTML tags, cryptographic algorithms, commands, and brands cause 20.16% of the false positives because the classifier confuses them with obfuscated strings. For instance, the brand name of the LG smartphone `LG G8X ThinQ` can be confused with an obfuscated string.

Observation 22 *Given this analysis, we identified short strings, hash sums, and the mix of foreign languages with short brand names as root causes for false positives and false negatives (**RQ3**).*

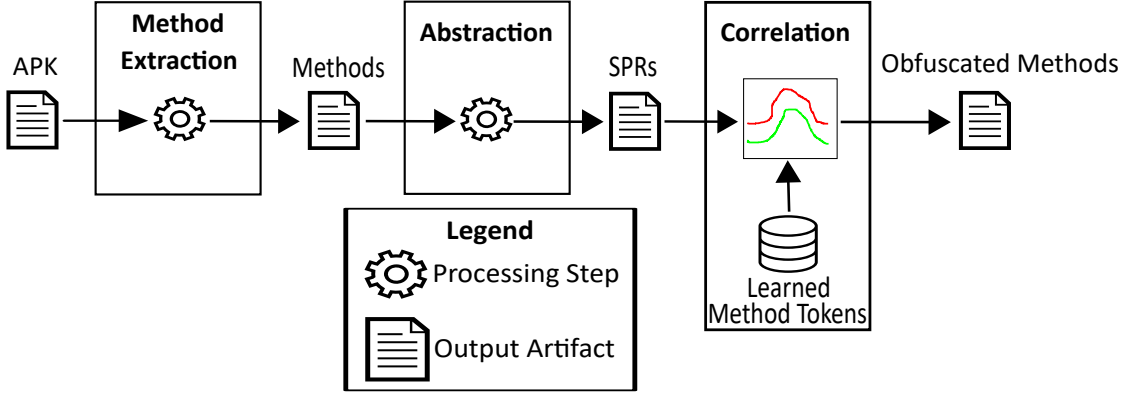
While the *String Classifier* identifies obfuscated strings represented by string constants, we introduce in the next section our approach that can identify obfuscated strings hidden in other data structures. For this purpose, it identifies deobfuscation logic that processes such strings.

9.4. Method Classifier

Inspired by the statistical text analysis [cry19], we assume that deobfuscation methods use specific instructions more often than other methods. Statistical analysis of text identifies English sentences by the high number of 'e' characters in it. Likewise, based on the observations of deobfuscation logic (cf. Chapter 8), we assume that it uses specific instructions more frequently than other logic. One example of such an instruction is `XOR`.

In order to compare the instructions of two methods, we build our *Method Classifier*, whose process is shown in Figure 9.2. The process extracts all methods, abstracts their content to SPR (cf. Section 3.2.1), and correlate the distribution of the SPR tokens with the ones of known deobfuscation methods.

In detail, we extract all instruction sets used in the deobfuscation methods, which we identified in Chapter 8, and use these instruction sets to identify the usage of obfuscated strings hidden in other data structures. After the extraction of all instruction sets per method, we transform the instructions into the *Structure-preserving Representation (SPR)* described in Section 3.2.1 to compare the resulting tokens with those of known deobfuscation methods. This representation preserves the structural tokens of a method's instructions but abstracts away information that gets changed in obfuscated code and would produce noise for the classification. E.g., we remove all name and type-information that does not occur in the Android standard library. Additionally, we identified during the analysis of the token distributions that store- and load-tokens of non-array variables are distributed randomly. As a consequence, we removed them from our considerations.

Figure 9.2.: Process of *Method Classifier* to identify deobfuscation logic

Due to these changes, the representation is robust against most of the obfuscation techniques described in Table 3.4 except the *Code Restructuring* and *Hide Functionality*. Using only the SPR, we cannot handle the two exceptions. However, we handle *Code Restructuring* by defining a clear structure and an order for the SPR-tokens. For the order, our *Method Classifier* sorts all SPR-tokens by their names and counts the occurrence of each token. We extract this structure for each deobfuscation method that was identified in Chapter 8. Afterward, the structures are stored in a database to compare the token distribution of these structures with the ones of a method under analysis.

```

1  byte[] b(byte[] p1, byte[] p2) {
2      byte[] r;
3      try {
4          Key key = null;
5          DESedeKeySpec des = new DESedeKeySpec(p1);
6          key = SecretKeyFactory.getInstance("desede").generateSecret(des);
7          Cipher cp = Cipher.getInstance("desede/ECB/PKCS5Padding");
8          cp.init(Cipher.DECRYPT_MODE, key);
9          r = cp.doFinal(p2);
10     } catch (Exception exception) {
11         r = null;
12     }
13     return r;
14 }

```

Listing 9.1.: Java method to decrypt Triple DES.

Listing 9.1 shows an example method that uses the DESede (Triple-DES) algorithm [Cry20] to decrypt a byte array that was encrypted using the same algorithm. This method takes a decryption key and an encrypted string as a byte array (i.e., parameters `p1` and `p2`) and returns the decrypted string using an additional byte array (line 13). Given the bytecode of this method, the *Method Classifier* abstracts the method using the *SPR* and counts each token occurrence in it.

Table 9.5 shows each *SPR*-token and its occurrence constructed from the method

9. Detection of Obfuscated Strings

Table 9.5.: Feature list for the Triple DES decryption method in Listing 9.1

Token	# of Token	Comment
<init>	1	new of DESedeKeySpec
along	1	introduced by goto in exception
byte[]	6	3 signature, 1 DESedeKeySpec, 2 doFinal
const	2	null for key & exception
doFinal	1	
dup	1	new of DESedeKeySpec
generateSecret	1	
getInstance	2	
if	1	introduced by goto in exception
init	1	
int	2	signature & Cipher.init
invoke	6	5 calls & new of DESedeKeySpec
java/lang/String	2	
java/security/Key	1	
java/security/spec/KeySpec	1	
javax/crypto/Cipher	4	
javax/crypto/SecretKey	1	
javax/crypto/SecretKeyFactory	3	
javax/crypto/spec/DESedeKeySpec	2	
ldc	2	
load	8	omitted for correlation
new	1	
return	1	
store	7	omitted for correlation
void	2	1 by <init> and 1 by init

in Listing 9.1. Since Listing 9.1 shows only the Java source code, we describe in the following all tokens that result from the bytecode to SPR transformation:

- The <init>, dup, and new tokens are constructed for each call of the new keyword in Java.
- The JVM handles all temporary variables using store- and load-instructions that we omit for the comparison.
- Branch tokens such as if replace all conditional and unconditional jumps. These tokens are followed by the direction of the jump. For instance, along is used for forward, and back for backward jumps.
- Each string constant is replaced by an ldc token and each constant value such as null is replaced by a const token.

- Finally, we replace all method calls by an `invoke` token.

Using all deobfuscation methods in the database, *Method Classifier* compares each of them with each app method. For the comparison, it calculates the Spearman correlation [MS04] between the tokens of the deobfuscation method and the method under analysis. It does not use all tokens for the correlation, but only the ones that the method under analysis shares with the method in the database. The Spearman algorithm [MS04] calculates the correlation between two distributions, even if they are not normally distributed. If one method correlates at least 85% (very strong correlation) with a known one, we consider it as a deobfuscation method.

Using this procedure, we identify even inlined deobfuscation logic because we correlate only the intersection of tokens from both methods. Therefore, if the deobfuscation method is inlined into another method, we extract only the tokens that are relevant for our comparison.

9.5. Evaluation

For the evaluation, we investigated the following research questions:

RQ1: How precise/accurate is the *Method Classifier* in identifying variations of known deobfuscation logic?

RQ2: How precise/accurate is the *Method Classifier* compared to the state-of-the-art deobfuscation method detectors?

RQ3: What kind of methods are falsely classified by the *Method Classifier*?

In order to answer these questions, we conducted three experiments. For the first experiment, we obfuscated apps using *DexGuard* [Dex19c] and *ZKM* [Zel19] that produce variations of their deobfuscation logic. Afterward, we execute *Method Classifier* on the obfuscated apps and measure the precision, recall, and F_1 -measure of it. The second experiment compares the outputs of two deobfuscation method detectors with the output of the *Method Classifier*. Lastly, we analyze methods that were falsely classified by our *Method Classifier* to identify potential issues for future work.

We performed our experiments using a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The experiments were executed on the server using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory.

9.5.1. Identification of Deobfuscation-Method Variations

Method Classifier's primary purpose is to locate deobfuscation schemes that represent obfuscated strings as other data structures. As reported in Chapter 8, only two tools produced variations of their deobfuscation logic in our analyzed data set (cf. BA in Table 8.1). For that reason, we use these tools to generate variations of their obfuscation schemes. Nevertheless, this experiment not only uses the representations that identify

9. Detection of Obfuscated Strings

the generated variations but also all other representations that were identified in Chapter 8. If we used only the representations that handle other data structures, this would lead to a bias towards lower false-positive occurrences. This bias can occur because the usage of fewer representations can lead to less falsely classified methods.

As mentioned above, we use two tools to generate variations of deobfuscation methods. In the following, we describe the different variations used by these tools. First, the tools vary the obfuscation keys by using random numbers. Second, they permute the order of formal parameters or change the method’s signature. Third, they alter the position of code blocks, whose execution order does not matter. Finally, they inline the deobfuscation logic based on the context of the string usages. For instance, if a class contains only one string usage, one tool inlines the deobfuscation logic at the string usage site. In other cases, the tool places this logic in a separate method.

To prepare our variation experiment, we applied both tools on 1,879 apps from the F-Droid data set (cf. Section 9.3). However, we were only able to generate 2,127 obfuscated apps, creating at least 1,000 apps for each obfuscator¹. The deobfuscation methods in the resulting obfuscated apps constitute our ground truth to evaluate our classifier’s effectiveness on variations of known obfuscation schemes. However, the extraction of these methods is not straightforward since obfuscators do not provide explicit information.

Nevertheless, since we obfuscated the apps on our own, we can extract this information for each app using the produced mapping files provided by the obfuscators. These mapping files enable app developers to find all original names in the source code for crash reports if they use obfuscated names. Consequently, if an obfuscated app contains methods or fields with no entry in its corresponding mapping file, the methods contain deobfuscation logic.

While the added methods contain the extracted deobfuscation logic, the methods with inline logic are unknown. However, we can identify the inline logic using the added fields by obfuscators. These fields contain all obfuscated strings in an array, and the deobfuscation logic accesses these fields. As a result, we constitute our ground truth by adding all new methods and all methods that access the newly added fields.

Given the mapping files, we obtain for our ground truth a list of 144,190 methods that contain deobfuscation logic, either in a separate method or inlined into previously existing methods.

To evaluate our classifier’s effectiveness, we measure its precision, recall, and F_1 -measure on the methods in the ground truth. Table 9.6 shows the comparison of this list with the *Method Classifier*’s output.

Observation 23 *Method Classifier identifies the analyzed variations of deobfuscation logic with a precision of 99.66%, a recall of 97.42%, and a F_1 -measure of 98.53%. With these results, we conclude that our classifier is very accurate for variations of logic, missing only a few methods with deobfuscation logic (RQ1).*

¹We could not obfuscate every app due to version incompatibilities between obfuscators and APKs.

Table 9.6.: Results of the variation experiment

Measure	Percentage
Precision	99.66%
Recall	97.42%
F_1	98.53%

9.5.2. Comparison With Other Detectors

In this section, we compare the deobfuscation method detectors of *Dex-Oracle 1.0.5* [Dex20] and *JMD 1.61* [Jav20a] against our *Method Classifier*.

In the last experiment, we identified all deobfuscation methods to compose a ground-truth. However, the identification of deobfuscation methods in all obfuscation schemes was impossible because of missing mapping files.

Since we cannot compose a ground truth for comparing the precision and recall of all approaches, we compare all approaches using an alternative notion of precision and recall, which uses the consensus metrics outlined by Lamiroy et al. [LS11]. The research community for graphic recognition uses these consensus metrics to compare various tools if no ground truth is available. However, before calculating the precision and recall based on the consensus metrics, we have to meet two requirements.

The first requirement demands that the output of each examined tool is treated equally. Since it is unknown to what extent the individual tools are accurate with the output of their calculations, this requirement ensures that each tool has an equal share of the potential truth. For instance, if five tools are compared, and only one tool reports a finding, it has a 20% chance to be true.

We can fulfill the first requirement for our evaluation scenario without any further efforts since we do not know which tool produces the most accurate results and therefore have to treat the outputs of these tools equally.

The second requirement demands that the analyzed data has to be equally distributed and equally probable. This requirement states that if we have a data set with two unique entries and one of the entries is present 50 times in our data set, the other must also be present 50 times to have an equal distribution. For an equal probability of entries, one needs to ensure that each entry represents a hit as probable as a miss.

To meet the second requirement, we analyzed the method distribution of multiple apps and identified that numerous methods occur more frequently across apps than others. For instance, multiple library methods can occur in different apps, but the methods of the main-app code occur most likely only in the one app that defined it in its code. All methods that appear more frequently have a higher distribution and a higher probability. Since we want to have a data set in which all methods are equally distributed, we filter out all methods that exist multiple times in the data set so that only one method of each kind remains in the data set. For the filtering, we use a cryptographic hash to build a signature of a method's bytecode and use only a single occurrence of this signature for our comparison.

9. Detection of Obfuscated Strings

To get an equal probability of deobfuscation methods and other ones, we use external knowledge of code and the tools under analysis. First, we exclude all methods that contained no strings before the obfuscation, and no tool classified them as deobfuscation methods. The remaining methods were classified as deobfuscation methods by at least one tool and most likely contained deobfuscation logic. Finally, to get our evaluation data set, we combined all unique methods from the set of potential deobfuscation methods with the same number of methods that contained no strings before the obfuscation.

Using the single occurrence of each method and the split into potential deobfuscation methods and non-deobfuscation methods, we achieve an equal distribution and an equal probability as demanded by the second requirement.

For the comparison of all tools, we randomly chose 1,000 obfuscated apps from the F-Droid data set described in Section 9.3. Afterward, we performed the above-described procedure and acquired 47,255 potential deobfuscation methods and the same number of unique methods randomly chosen from the set of excluded methods. For the comparison, we used the precision consensus metric in Equation 9.1 and for the recall the one in Equation 9.2 [LS11].

$$Precision(S_k) = \frac{\sum_{i=1\dots d} P(\delta_i) S_k(\delta_i)}{\sum_{i=1\dots d} S_k(\delta_i)} \quad (9.1)$$

$$Recall(S_k) = \frac{\sum_{i=1\dots d} P(\delta_i) S_k(\delta_i)}{\sum_{i=1\dots d} P(\delta_i)} \quad (9.2)$$

$$P(\delta_i) = \frac{1}{s+2} \sum_{k=1\dots s, \perp, \top} S_k(\delta_i) \quad (9.3)$$

The variable S_k defines in our scenario one of k tools under analysis that given δ_i and one of d methods returns either 1 if the method contains deobfuscation logic or 0 otherwise. $P(\delta_i)$ defines the consensus probability in Equation 9.3 [LS11] that the given method is truly a deobfuscation method. Whereas S_{\top} defines the tool that always returns 1 and S_{\perp} always returns 0 for any given method.

Next, given our evaluation data set, we calculate for each tool the consensus metrics for precision, recall, and the resulting F_1 -measure, and show the results in Figure 9.3.

While the results do not reflect the actual precision and recall for methods that contain deobfuscation logic, we can use the results to compare the tools. As shown in Figure 9.3, *Method Classifier* has a precision of 43.16%, which is 12 percentage points lower than the precision of the best tool (*JMD* detector). However, the recall of the *Method Classifier* is 60.34%, which is 44.8 percentage points higher than the recall of the second-best tool (*Dex-Oracle* detector).

Since these results do not reflect the real distribution of methods, we also executed all tools on all methods of the 1,000 obfuscated apps. The results show that all tools filtered out 945,315 unique methods, which do not contain deobfuscation logic and amount to 95% of the methods in all apps. Since these methods contained no string before the obfuscation, the *Method Classifier* achieves a true negative rate of at least 95%.

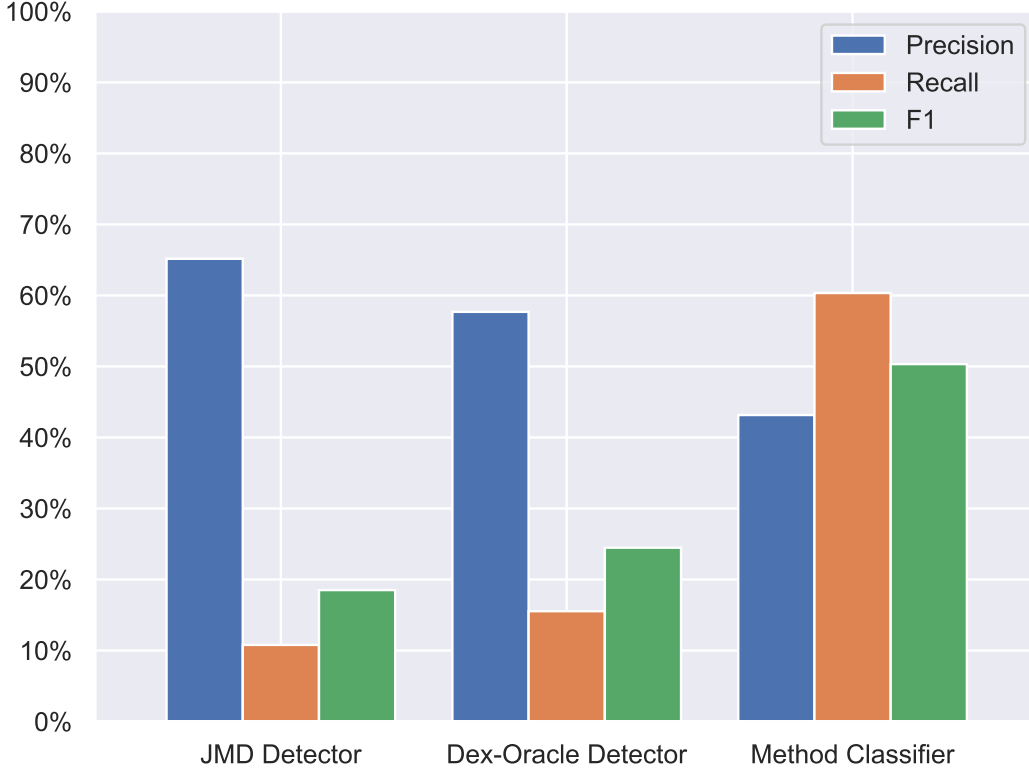


Figure 9.3.: Consensus precision, recall and F_1 -measure for all tools

Observation 24 *Given the filtered out 95% of the non-deobfuscation methods, the Method Classifier identifies more than twice as many deobfuscation methods as identified by any other tool with a precision loss of only 27.8 percentage points (RQ2).*

To explain the differences in precision and recall between the two previous experiments, we present, in the next section, our manual analysis of false positives and false negatives.

9.5.3. Falsely Classified Deobfuscation Methods

Given all unique methods from the previous section, we analyze in this section, only the ones that potentially contain deobfuscation logic. Since all tools filtered out 95% of the unique methods, we have 47,255 methods left for our analysis. However, a detailed analysis of all these methods is not feasible. Consequently, we excluded all methods that are either flagged by all three tools as deobfuscation methods or are identified using the mapping procedure from Section 9.5.1. From the remaining methods, we randomly chose 100 methods that our *Method Classifier* flagged as deobfuscation methods and manually identified 41 false positives.

To analyze false negatives, we used all methods that were not flagged as a deobfuscation method by the *Method Classifier* and excluded all methods that contained no string

9. Detection of Obfuscated Strings

before the obfuscation. Only 75 methods remained for our analysis, and we identified only four false negatives.

Table 9.7.: Sample of false-classified methods

Description	Category	Percentage	Number of Methods
Calls Deobfuscation Method	FP	69.49	41
Previous Usage of Byte Arrays	FP	30.51	18
Variations of Known Obfuscation Schemes	FN	76.67%	23
Split of Deobfuscation Logic	FN	23.33%	7

A detailed analysis of false positives and false negatives separates each of them into two categories, which we show in Table 9.7. The analysis of the chosen 100 methods revealed that 59 methods were false positives with 69.49% of these methods contain calls to deobfuscation methods, and the remaining 30.51% contain byte arrays used to encode text to a portable document format (PDF) or implement low-level communication layers via data-grams. The usages of byte arrays add noise to the measured token distribution and cause a higher correlation between the method under analysis and our set of known deobfuscation methods, resulting in false positives.

The analysis of false negatives shows that out of 75 methods, 30 methods are indeed false negatives. The *Method Classifier* has not identified seven of the methods because their deobfuscation logic is split into multiple methods, and our classifier identified the other methods of the logic. Therefore, these methods would not cause missing obfuscated strings. Nevertheless, the remaining 76.67% of the 30 methods are indeed false negatives caused by variations of known obfuscators.

Observation 25 *Given these results of all experiments, we conclude that the Method Classifier has a low false-negative rate and is suitable to identify all analyzed deobfuscation methods. The analysis shows that most of the false positives are caused by integrating calls to deobfuscation methods (RQ3).*

9.5.4. Discussion

In the previous experiments, the *Method Classifier* identified variations of deobfuscation methods with a precision of 99.66% and a recall of 97.42%. Compared with other tools, it identified deobfuscation methods with a precision of only 43.16% and a recall of 60.34%. However, many methods that are flagged as false positives manipulate an obfuscated string before propagating it to a deobfuscation method. Consequently, these methods are no real false positives but belong to a deobfuscation logic that is split across multiple methods. Since the other tools cannot detect inlined deobfuscation logic, they flag methods containing such logic as non-deobfuscation methods and cause a higher probability of a false positive.

The same applies to false negatives since some obfuscators move the entire code into another method and add a delegate method with a deobfuscation-method signature.

Consequently, the other tools flag it as a deobfuscation method, while the *Method Classifier* does not. It flags only the methods that contain the actual deobfuscation logic.

As a result, the consensus metrics for precision and recall under-represent the identification capabilities of *Method Classifier* but show that *Method Classifier* prefers recall over precision compared to other tools.

9.6. Threats to Validity

We identified the following threats to the validity of our experiments. If an obfuscator adds random words to a string, it can eventually evade the detection using the *String Classifier* because the proportion of the non-obfuscated content will increase. However, for this technique to be effective, more than half of a given string need to consist of non-obfuscated words because the *String Classifier* decides at this threshold that the string might be non-obfuscated. During our analysis of obfuscation techniques, we never found more than one dictionary word in an obfuscated string.

Another threat could be that a new obfuscator manipulates a string that does not share any commonalities with known techniques. In order to identify this new technique in the future, we would need to extend the approach. Nevertheless, we do not need to train our classifiers from scratch because we can train an additional REP Tree for this technique and consider a string as obfuscated, as long as at least one of the REP Trees classifies it as such. Similarly, we can add the SPR of the new technique to the list of our *Method Classifier* to adapt its classification.

Finally, if an obfuscator hides all obfuscated strings in an encrypted class, neither the *String Classifier* nor the *Method Classifier* can detect these strings. Additionally, if obfuscators use probing [XQE16] to identify vulnerabilities of classifiers that identify obfuscated strings, we would need to identify features that are not affected by probing. However, currently, no other string obfuscation detector handles such techniques.

9.7. Conclusion

In this chapter, we introduced two approaches that complement each other to identify obfuscated strings in apps. The first approach, *String Classifier*, directly analyzes strings with a classifier that identifies obfuscated strings based on their characteristics. The second approach, *Method Classifier*, compares methods with the ones that contain known deobfuscation logic to identify obfuscated strings hidden in other data structures.

String Classifier analyzes strings for words from different natural languages and uses different features from cryptanalysis, known formats, and word characteristics to identify obfuscated strings. Whereas, the *Method Classifier* extracts a specific representation of each method and compares the instructions of it with known deobfuscation logic using the correlation between the occurrences of different instructions.

Since most other tools cannot detect obfuscated strings on an individual basis, we evaluated the *String Classifier*'s effectiveness only on our data set without comparing

9. Detection of Obfuscated Strings

it with other tools. However, with a F_1 -score above 94%, the *String Classifier* is well suited to identify obfuscated strings in the wild.

While for the *String Classifier*, we had a well-defined data set and no comparison tools, we have for the evaluation of the *Method Classifier* tools for the comparison, but no well-defined data set. The collection of a well-defined data set was impossible because many obfuscators do not document the methods which they use for deobfuscation. Consequently, we used a technique from graphic recognition that assesses multiple tools without the necessity of a ground truth.

After establishing the necessary conditions for the usage of this technique, we could analyze the relations between each tool's precision and recall. The results indicate that the *Method Classifier* identifies most of the deobfuscation logic.

In this chapter, we evaluated the *String Classifier*, and the *Method Classifier* individually. In the next chapter, we use both of the classifiers to evaluate their performance in the wild. For instance, we show that the usage of both classifiers yields the best coverage of obfuscated strings in the wild.

10. String Deobfuscation

In this chapter, we introduce *StringHound*, an analysis that combines the detection approaches of the previous chapter with a light-weight slicing technique that extracts and executes the deobfuscation of a string.

StringHound utilizes the fact that the entire code necessary to deobfuscate a string resides in the deployed application [WHA⁺18]. We use this code by applying slicing to retrieve all necessary instructions that compute the original deobfuscated string. Weiser [Wei81] introduced slicing in 1981 and since then various other approaches [Cha17, RAMB16, LLTX17, MW17, ZWWJ15, GZZ⁺12, HUHS13] use it.

The usage of Weiser’s slicing algorithm on its own does not solve the entire deobfuscation problem. Our approach faces three challenges to deobfuscate strings. The first challenge is the increased complexity of the obfuscated code. Obfuscation tools produce more complex code by transforming the syntax while keeping the semantics unchanged. Consequently, all algorithms that are only processing not obfuscated bytecode are not suited to handle the complexity produced by obfuscators. For instance, in contrast to standard compilers, obfuscators can produce irreducible control-flow graphs. These graphs may contain multiple entry points to loops by using `goto` statements.

The second challenge consists of countermeasures that are implanted into the bytecode and checked during runtime to evade simple string inspections. For instance, as described in Chapter 8, countermeasures check whether an app is executed in a false environment.

While the first two challenges focus on code transformations, the last one focuses on the scope of the deobfuscation analysis. The number of analyzed strings directly affects the runtime of *StringHound*. Therefore, the more strings *StringHound* can filter out that do not represent an obfuscated-string usage, the less time it spends with the deobfuscation. While *StringHound* can reduce the number of analyzed methods by leveraging the identification capabilities of our classifiers, the reduction of string usages in these methods requires detailed knowledge about the capabilities of obfuscators. Since their deobfuscation logic cannot manipulate methods contained in the Android class library, we reduce the number of string usages by analyzing only code structures that do not direct constant strings into these methods of the Android class library.

Given such a code structure, *StringHound* primarily performs backward slicing interleaved with forward-phases if necessary to get an executable slice. For example, the creation of new objects in Java bytecode requires forward phases during the slicing process. In contrast to approaches that reconstruct high-level code structures such as for-loops, *StringHound* directly processes low-level bytecode instructions. As a result, it avoids issues caused by obfuscators, such as irreducible control-flow graphs.

In addition to the processing of low-level bytecode, *StringHound* circumvents dynamically invoked countermeasures that are applied by current obfuscation tools. For in-

10. String Deobfuscation

stance, *StringHound* does not change the information in stack traces that are checked by known deobfuscation logic.

We evaluated *StringHound* and four available state-of-the-art deobfuscation tools [Dex20, sim20, Jav20a, Dex19b] by applying them to a set of apps that we obfuscated with 21 different techniques. The evaluation shows that *StringHound* yields significantly better results than the other tools. We also applied *StringHound* to four sets of benign and malicious real-world apps: (a) a random sample of 100,000 apps, (b) popular apps based on AndroidRank’s Top 500 [And19], (c) malware from Contagio [Con19b], and (d) apps from the Google Play store in 2018 classified as malicious by VirusTotal [vir19]. The classifiers in Chapter 9 were vital in enabling a study of more than 100,000 apps by using them to filter out apps that do not contain any obfuscated strings to avoid unnecessary slicing and deobfuscation steps.

Our study shows that apps contain string obfuscation at least 12 times more often than claimed by previous studies [Don18, Mir18, WR17]. We give insights into the found obfuscated strings that we categorize by a list of common security-related patterns. Besides expected results, such as obfuscated URLs and commands in malware sets, we surprisingly found that 76% of the 100,000 apps contain obfuscated strings. An in-depth analysis revealed that ad libraries integrated into apps contain several obfuscated strings. Moreover, we identified two apps in the Top 500 set that conceal suspicious behavior through string obfuscation. They collect sensitive information from a user’s mobile phone, such as call logs and location information, to build a user profile for tracking. Furthermore, they also check for the `SuperUser.apk` that grants root access to the mobile phone. These apps are installed on over 20 million devices and are not flagged as malicious by VirusTotal [vir19].

In the next sections, we discuss state-of-the-art string deobfuscators in Section 10.1, our approach *StringHound* in Section 10.2, a detailed evaluation of *StringHound*’s capabilities in Section 10.3, threats to the validity of our experiments in Section 10.4, and draw a conclusion in Section 10.5.

10.1. State-of-the-Art String Deobfuscators

In this section, we discuss all deobfuscators, even though not all were available for empirical comparison. First, we discuss *Dex-Oracle* [Dex20], *Simplify* [sim20], *JMD* [Jav20a], and *DEX2JAR* [Dex19b] which we also use for empirical evaluation. Furthermore, we examine *CredMiner* [ZWWJ15], *Harvester* [RAMB16], *ARES* [BP18], *TIRO* [WL18], *FlowSlicer* [MW17], *SAAF* [HUHS13], *Tiger* [CYL⁺17], and *AGRIGENTO* [CFL⁺17], which are not publicly available or can only deobfuscate strings in a restricted setting.

Dex-Oracle [Dex20] searches for deobfuscation methods and executes them in an emulator as a part of an app. As described in Section 9.1, it uses fixed method signatures to filter the app code for deobfuscation methods. *Dex-Oracle* misses inlined deobfuscation code and signature variations of deobfuscation methods. For instance, we found in our study of `cn.pro.sdk` (cf. Chapter 8) inlined deobfuscation logic produced by a known

obfuscator. Moreover, it assumes that all constant values needed for the execution of the deobfuscation method are provided directly before the specific call of that method, while the latter can also be the result of field accesses or other computations.

Simplify [sim20] applies semantic-preserving transformations to optimize an app's code. Example transformations are constant propagation and dead code removal. To enable transformations, it executes each method on a virtual machine sandbox and returns a graph with all possible register and class values for every execution path.

Simplify can only deobfuscate strings that do not depend on any state. In such cases, the constant propagation of *Simplify* can uncover hidden information. Due to two reasons, *Simplify*'s support for string deobfuscation is limited. First, it is not able to decide which operations or values are necessary to deobfuscate a string. Second, it cannot handle deobfuscation methods that use keys stored in fields.

JMD [Jav20a] re-implements deobfuscation logic of known obfuscators [Zel19, All19, Das19] to execute it with directly-propagated constants. *JMD* extracts these constants from identified immediate callers of the known deobfuscation methods. After executing the deobfuscation logic, *JMD* replaces calls to deobfuscation methods by revealed strings.

However, *JMD* does not consider field accesses or other ways to retrieve the propagated values. It identifies obfuscated strings by searching for specific loading instructions (LDC). Consequently, *JMD* misses almost all obfuscated strings, which would be produced by the obfuscation techniques from Chapter 8 because the techniques load them by using another instruction. Additionally, as described in Section 9.1, *JMD* uses a fixed set of method signatures without considering variations or inlining of deobfuscation logic, which also leads to significant misses of obfuscated strings.

DEX2JAR [Dex19b] transforms Dalvik bytecode to Java bytecode. However, it also has a sub-module that executes methods with a certain signature for deobfuscation purposes. Similar to our approach, *DEX2JAR* executes code in the underlying JVM. However, *DEX2JAR* needs the user to identify and provide the deobfuscation methods. Additionally, *DEX2JAR* has similar drawbacks as *JMD* and *Dex-Oracle*, such that it assumes all inputs for the deobfuscation-method call directly above it. Unfortunately, none of the obfuscation techniques that we surveyed in Chapter 8 matches these conditions.

CredMiner [ZWWJ15] extracts and deobfuscates developer credentials (e.g., usernames, passwords) from Android apps by performing backward slicing starting from known methods that take credentials as parameters. Since *CredMiner*'s extracted slices are not directly executable, they miss constructor calls of objects. To make them executable, *CredMiner* creates mock objects and uses a custom runtime environment. Since the mocked objects cannot reproduce the whole behavior of the real objects, *CredMiner* cannot handle advanced countermeasures that, for instance, rely on the shape of the stack. Furthermore, *CredMiner* uses a manually crafted list that needs to be maintained to identify current methods that use user names and passwords as parameters. Lastly,

10. String Deobfuscation

CredMiner cannot handle obfuscated strings in char or byte arrays since its custom runtime environment models only Java’s String API.

Even though *CredMiner* is highly related to our work, we were not able to acquire it for an empirical comparison even in the restricted setting of deobfuscating credential strings because it is not publicly available.

Harvester [RAMB16] implements a combined static and dynamic analysis to extract obfuscated runtime values from Android malware. It uses *SuSi* [RAB14] to identify sources and sinks needed for *Harvester*’s slicing technique. Since *SuSi* relies on the names of methods and classes, it is not suited to identify name-obfuscated methods.

Harvester starts with the identified sinks and performs backward slicing to identify `if` statements that potential malware may use to prevent the exposure of its payload. *Harvester* rewrites the identified statements so that it can systematically enumerate all paths to the sink while the Android simulator executes the sliced app. Using this procedure, *Harvester* circumvents countermeasures relying on `if` statements. However, because *Harvester* does not preserve stack information, it cannot handle the usage of this information as a deobfuscation key.

Since *Harvester* is not publicly available, an empirical comparison is not possible for the kind of obfuscation schemes that can be handled by *Harvester*. Even if *Harvester* would be available, it is not suited to deobfuscate all strings because it cannot detect deobfuscation logic.

ARES [BP18] identifies and executes unexplored paths to malware payload hidden by countermeasures. To identify these paths, *ARES* starts with methods specified in a static list of sources and sinks and performs backward slicing. *ARES* tries to identify `if` statements that prevent the exposure of potential malicious payload by using environmental values. As *Harvester*, *ARES* rewrites the identified statements to systematically enumerate all paths to the sink while the Android simulator executes the sliced app.

Given this procedure, the approach circumvents countermeasures relying on `if` statements. However, because *ARES* does not preserve stack information, it cannot handle the usage of this information as a deobfuscation key. Additionally, due to the usage of a static list, *ARES* is not suited to identify obfuscated methods used for deobfuscation because this list relies on class and method names.

While *ARES* is publicly available, it is not suited to deobfuscate all strings because it cannot detect deobfuscation methods.

TIRO [WL18] applies instrumentation for Android apps to identify runtime obfuscation that hides data or the execution order of code. For the identification, it instruments a specified list of methods, uses fuzzing to execute these methods with various input values, and observes the resulting outputs. *TIRO* uses *IntelliDroid* [WL16] to identify call paths to the specified list of methods in the app. Nevertheless, *IntelliDroid* is not suited to identify obfuscated methods used for the deobfuscation of strings because it relies on method and class names. After detecting call paths, *TIRO* instruments the entire app

and the Android runtime to extract values during execution. However, for the instrumentation, *TIRO* still needs a static list of methods and therefore suffers from the same drawbacks as *ARES*.

Since *TIRO* is not publicly available, an empirical comparison is not possible for the kind of obfuscation schemes that can be handled by *TIRO*. Even if *TIRO* were available, it is not suited to deobfuscate all strings because it cannot detect deobfuscation methods.

FlowSlicer [MW17] combines static and dynamic analyses to detect sensitive information leaks in Android applications. The static analysis locates source and sink methods related to potential information leaks. By backward-slicing, the approach determines if the information passed to a sink method potentially originates from a relevant source. If so, *FlowSlicer* instruments the code such that it can track and observe information flows at runtime. To track the information flows, *FlowSlicer* executes the instrumented application on an Android simulator. This procedure enables *FlowSlicer* to check if an app indeed leaks sensitive information.

While not designed for string deobfuscation, we could use *FlowSlicer* for that purpose, given appropriate sources and sinks. However, it executes only the original code with the instrumentation to track information flows. Hence, even the simplest countermeasures are sufficient to hinder the deobfuscation of a sensitive string. Furthermore, it is only possible to get a deobfuscated string if we can trigger the path to a specific sink. However, to trigger such paths, we would require specific test cases or user inputs.

Since *FlowSlicer* is not publicly available, an empirical comparison is not possible for the kind of obfuscation schemes that can be handled by *FlowSlicer*. Even if *FlowSlicer* would be available, it is not suited to deobfuscate all strings because it cannot detect deobfuscation logic.

The Static Android Analysis Framework (SAAF) [HUHS13] was designed to facilitate the analysis of Android malware. It is a generic slicing-approach that tracks the flow of constant values, in particular, string constants. *SAAF* uses use-def chains for its slicing to identify sources of the constants. *SAAF* uses interleaved backward forward-slicing to identify all relevant instructions but does not aim at extracting or executing the slice and, therefore, would not deobfuscate strings. It just identifies the sources related to their usage. As a consequence, users would have to perform the deobfuscation manually.

Since *SAAF* is not publicly available, an empirical comparison is not possible. Even if *SAAF* were available, we would need to perform the deobfuscation manually.

Tiger [CYL⁺17] generates a network-traffic signature to categorize apps as malicious or benign. *Tiger* constructs the signature by slicing all constant information belonging to a selected list of Android’s network methods. Afterward, it partially executes the slices to reduce the size of each signature. *Tiger* starts constructing the signature at a list of methods instead of automatically analyzing the app’s codebase. As a result, *Tiger* misses methods from new Android versions. Moreover, it misses reflectively called network methods when the targets are obfuscated strings. While the extraction of network

10. String Deobfuscation

signatures may reveal many obfuscated strings, *Tiger* cannot deobfuscate strings that are not related to networks.

Since *Tiger* is not publicly available, an empirical comparison is not possible. Even if *Tiger* would be available, it is not suited to deobfuscate all strings because it does not identify obfuscated strings.

AGRIGENTO [CFL⁺17], similar to *Tiger*, generates a network-traffic signature to categorize apps as malicious or benign. In contrast to *Tiger*, *AGRIGENTO* constructs a signature by instrumenting specific methods and executing the entire app multiple times to explore more and more constants in each iteration. It starts constructing the signature at a static list of methods instead of automatically analyzing the app's codebase. As a result, it misses methods in new Android versions. As *Tiger*, *AGRIGENTO* misses reflectively called network methods when the targets are obfuscated strings. While the extraction of network signatures may reveal many obfuscated strings, *AGRIGENTO* cannot deobfuscate strings that are not related to networks.

Since *AGRIGENTO* is not publicly available, an empirical comparison is not possible. Even if *AGRIGENTO* would be available, it is not suited to deobfuscate all strings because it does not identify obfuscated strings.

Discussion Given the five concepts of string obfuscation in Section 8.4, and the descriptions of the deobfuscators, we show in Table 10.1 the connections between the concepts and the approaches. The five concepts include the hiding of strings (Hide Strings), keys (Hide Keys), logic (Hide Logic), the data flow of the logic (Hide Data Flow), and the protection of the deobfuscation execution (Protect Execution). Additionally, we add the automatic identification of obfuscated strings to the table (Identification). The handling of all concepts varies between fully (+), partially (\pm), and not handled (-).

We observe multiple commonalities and discrepancies between different groups of approaches. First, *SAAF* and *DEX2JAR* cannot handle any of the obfuscation concepts. While *SAAF* requires the entire slicing, *DEX2JAR* needs only an extension with a string-obfuscation detector to enable deobfuscation. Second, the tools *Dex-Oracle* and *JMD* have similar capabilities to handle the string-obfuscation concepts. Nevertheless, each of them handles the identification of hidden strings differently.

Third, *Tiger* and *AGRIGENTO* have not only similar capabilities but also the same shortcomings to handle string-obfuscation concepts. They can only deobfuscate strings related to network functionality.

Fourth, besides the missing automatic identification of *ARES*, it shares all capabilities with *Harvester* and *TIRO*. While these tools do not handle hidden strings and logic, the tools can deal with the highest number of obfuscation concepts.

Last, the tools *Simplify*, *CredMiner*, and *FlowSlicer* have either a particular focus or are not able to handle advanced concepts of string obfuscation.

While all tools deal with different string obfuscation concepts, none of them can handle all of the concepts. As a result, we need an approach that does not only automatically

Table 10.1.: String obfuscation concepts vs. detection and deobfuscation approaches

Approaches	Automatic Identification	Hide Strings	Hide Keys	Hide Logic	Hide Data Flow	Protect Execution
<i>Dex-Oracle</i>	+	+	-	-	-	-
<i>Simplify</i>	-	±	-	-	-	-
<i>JMD</i>	+	+	-	-	-	-
<i>DEX2JAR</i>	-	-	-	-	-	-
<i>CredMiner</i>	+	-	+	-	-	-
<i>Harvester</i>	±	-	+	-	+	±
<i>ARES</i>	-	-	+	-	+	±
<i>TIRO</i>	±	-	+	-	+	±
<i>FlowSlicer</i>	-	-	±	-	±	-
<i>SAAF</i>	-	-	-	-	-	-
<i>Tiger</i>	-	±	±	±	±	-
<i>AGRIGENTO</i>	-	±	±	±	±	-

identifies obfuscated strings but also uses a specially targeted slicing technique. In the next section, we present *StringHound* that possesses both capabilities.

10.2. StringHound

StringHound processes bytecode in five steps. Figure 10.1 shows a high-level view of this process. To reveal obfuscated strings, we need to identify the methods that potentially use them. For locating usages of obfuscated strings, we use the *String Classifier* and *Method Classifier* from the Chapter 9. Given the output of the classifiers, we find the starting point for the slicing (slicing criterion) in the methods that contain the usage of the obfuscated strings. Next, we use a specially targeted slicing technique that computes all program statements that affect the state of a given slicing criterion. Finally, *StringHound* injects the slice into the execution context of the deobfuscation logic. Afterward, it executes the resulting slice to obtain deobfuscated strings. The injection of the slice into the context renders countermeasures introduced by analyzed obfuscators ineffective. Our detailed description of *StringHound* shows the design decisions taken to address the obfuscation schemes presented in Chapter 8.

10.2.1. Slicing Relevant String Usages

In order to extract and execute the logic from deobfuscation methods, we need to define a criterion from which *StringHound* starts the slicing. We define as slicing criterion (s_{crit}) any instruction within a set of selected methods, which we call candidate methods, that produces a string value. A method m is in the set of candidate methods if (a) it contains instructions that consume a char sequence as a parameter (method calls, but also field

10. String Deobfuscation

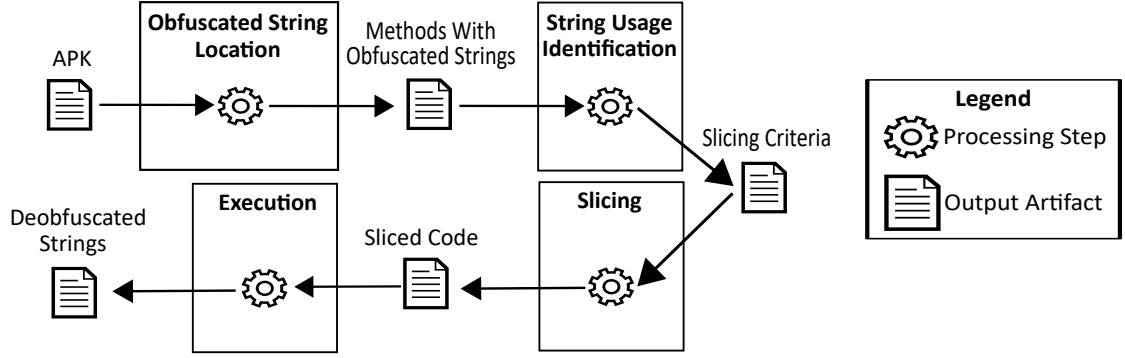


Figure 10.1.: Overview of *StringHound*'s process

writes, array stores, and return instructions), called *locations of interest (LoIs)*, and (b) satisfies one of following conditions: (i) the *String Classifier* found obfuscated strings in *m*, (ii) *m* calls a method *n* that the *Method Classifier* classified as a deobfuscation method, or (iii) *m* itself is classified as a deobfuscation method (inlined deobfuscation logic).

Since the classifiers identify neither LoIs nor slicing criteria directly, we have to search for them in the candidate methods. We use *OPAL* [EH14] to find all instructions that operate on values of type `CharSequence`, or a subtype thereof, in particular `java.lang.String`. We consider all expressions as s_{crit} that LoIs consume and may result in a string. Given a candidate method that contains LoIs, we identify all s_{crit} while ignoring constant string expressions. With all identified s_{crit} , we describe our targeted slicing in the following section.

10.2.2. Our targeted Slicing

Our slicing algorithm (cf. Figure 3) performs backward slicing [BG96, AARUJ86] with forward-phases to collect all instructions necessary for the execution of other relevant instructions. For instance, if the slice contains a *new* instruction, we collect their corresponding constructor invocation in forward-phases. Additionally, if several potential sources for a given string parameter are present, we start from each of them as a separate slicing criterion.

For example, Listing 10.1 shows two sources of *msg* (Line 2) corresponding to the two branches of the tertiary operator (Line 1), which load either `"US()"` or `"INT()"`. In such cases, *StringHound* would start the slicing process for each source.

```

1 String msg = simCountryIso().equals("US") ? US() : INT();
2 invoke("+01234", msg);

```

Listing 10.1: Example with two sources

Traditional slicing algorithms (cf. Binkley et al. [BG96]) inspired our technique that we adapted to our particular needs and implemented it using *OPAL* with definitions (cf. Aho et al. [AARUJ86]) of the functions defined in Table 10.2.

Given a method body along with its control-flow graph (CFG), a *LoI* and a slicing

Algorithm 3: Slicing algorithm

Input: m a method with a body
 I the instructions of the method m
 g the CFG of m where each $i \in I$
corresponds to one node $n \in N$ of g
 $LoI \in I$ the location of interest
 $s_{crit} \in I$ the slicing criterion

Output: $N_{slice} \subseteq I$

```

1  $N_{slice} := \{\}$ 
2  $W := \{s_{crit}\}$ 
3  $cd_{crit} := cd(s_{crit})$ 
4  $br_{LoI} := br(LoI)$ 
5 while  $W \neq \emptyset$  do
6    $currInstr := head(W)$ 
7    $W := W \setminus currInstr$ 
8   if  $currInstr \notin N_{slice}$  then
9      $N_{slice} := N_{slice} \cup \{currInstr\}$ 
10     $D := \{d \mid x \in use(currInstr) \wedge d \in ud(x, currInstr)\}$ 
11     $cd_{currInstr} := cd(currInstr) \setminus cd_{crit}$ 
12     $U := \{u \mid x \in def(currInstr) \wedge u \in du(x, currInstr)$ 
13       $\wedge u \in br_{LoI}\}$ 
14     $W := W \cup D \cup cd_{currInstr} \cup U$ 

```

criterion s_{crit} , our algorithm initializes the worklist W (Line 2 of Figure 3) with the slicing criterion s_{crit} . Afterward, it performs for each instruction in W (Line 6) that is not already part of the slice (Line 8) the following steps:

1. It adds the current instruction $currInstr$ to the slice (Line 9).
2. In the *backward phase* (Line 10), it determines the set D of all definition sites related to $currInstr$, i.e., D consists of instructions that initialize variables used by $currInstr$.
3. Also in the *backward phase*, it determines the set $cd_{currInstr}$ of instructions on which the current instruction is control dependent on (Line 11). From this set, it removes the instructions cd_{crit} that could prevent the execution of s_{crit} . This *backward phase* adds instructions to W that (in)directly affect s_{crit} . With this addition, the algorithm includes condition instructions that do not control the execution of the criterion itself. This step is required to, e.g., ensure that we add loops manipulating byte arrays to the slice. If the *backward phase* adds instructions that define a new reference-typed variable,

Table 10.2.: Definitions of helper functions for the algorithm

def	$Instr \rightarrow \mathcal{P}(Var)$	variables defined by an instruction
use	$Instr \rightarrow \mathcal{P}(Var)$	variables used by an instruction
du	$Var \times Instr \rightarrow \mathcal{P}(Instr)$	definition-use instructions
ud	$Var \times Instr \rightarrow \mathcal{P}(Instr)$	use-definition instructions
cd	$Instr \rightarrow \mathcal{P}(Instr)$	transitive control dependency instructions
br	$Instr \rightarrow \mathcal{P}(Instr)$	set of backwards reachable instructions

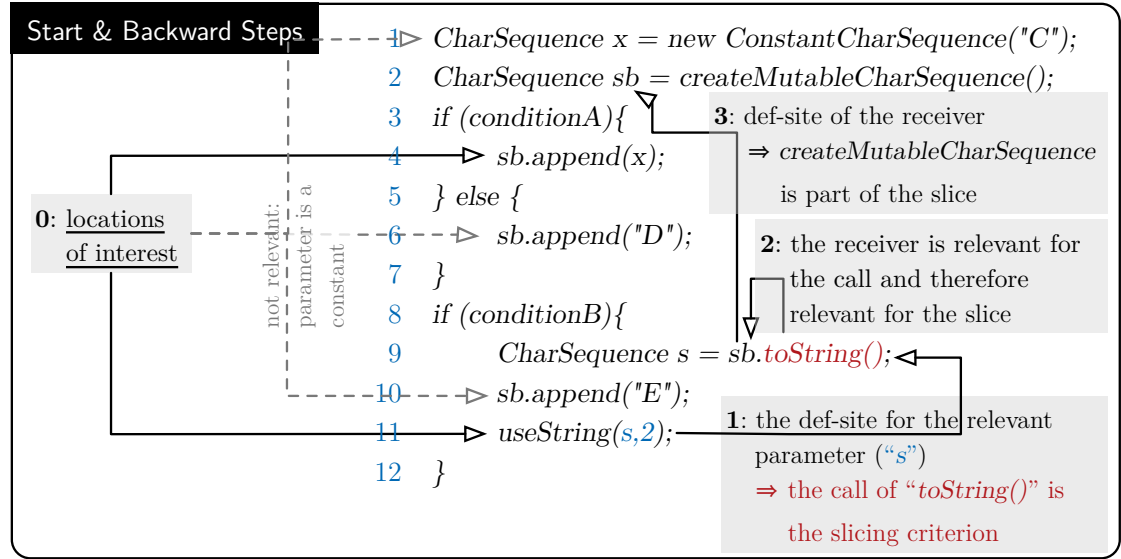


Figure 10.2.: Example of the slicing process for the parameter `s` of `useString`—Showing *LoIs* and first backwards step

i.e., an object, the algorithm performs an additional *forward phase* to include those instructions in W that potentially affect the state of the object after its initialization and which are relevant w.r.t. the *LoI*. Hence, the algorithm only adds instructions that are still backward reachable from the *LoI*.

4. In the *forward phase* (Line 12, 13), the algorithm determines the set of all instructions U that use a variable defined by $currInstr$ and which are backward reachable from the *LoI*. This phase includes all instructions that potentially mutate the state of the defined variable, e.g., filling an array with actual values or calling a method of the object.

5. In the last step (Line 14), W is updated with the following three sets of instructions. The set of instructions on which $currInstr$ is control dependent ($cd_{currInstr}$), those using the variable defined in it (U), and those initializing the variables used by it (D).

Given our slicing algorithm, the example in Figure 10.2 illustrates the process of determining s_{crit} and the algorithm's backward phase, divided into four steps. In step 0, the algorithm determines candidates for *LoIs* statements. Here, it selects the constructor in Line 1, the calls in the Lines 4, 6, 10, and 11. Given a *LoI*, the algorithm considers the definition sites (def-sites) of the instructions that load the *LoI*'s string parameters as slicing criteria (without including the *LoI* itself). For illustration purposes, we assume that the currently processed *LoI* is the call in Line 11. Its only string parameter `s` (the `int` parameter is ignored) is defined by the result of the call in Line 9. Hence, this call is our slicing criterion. On the contrary, we do not consider instructions that load string constants as slicing criteria, e.g., the call in Line 6 is a *LoI*, but the instruction that loads the string constant "D" is not a slicing criterion. The rationale behind this is that in this case, no deobfuscation can happen before reaching the *LoI*. In Figure 10.2 such

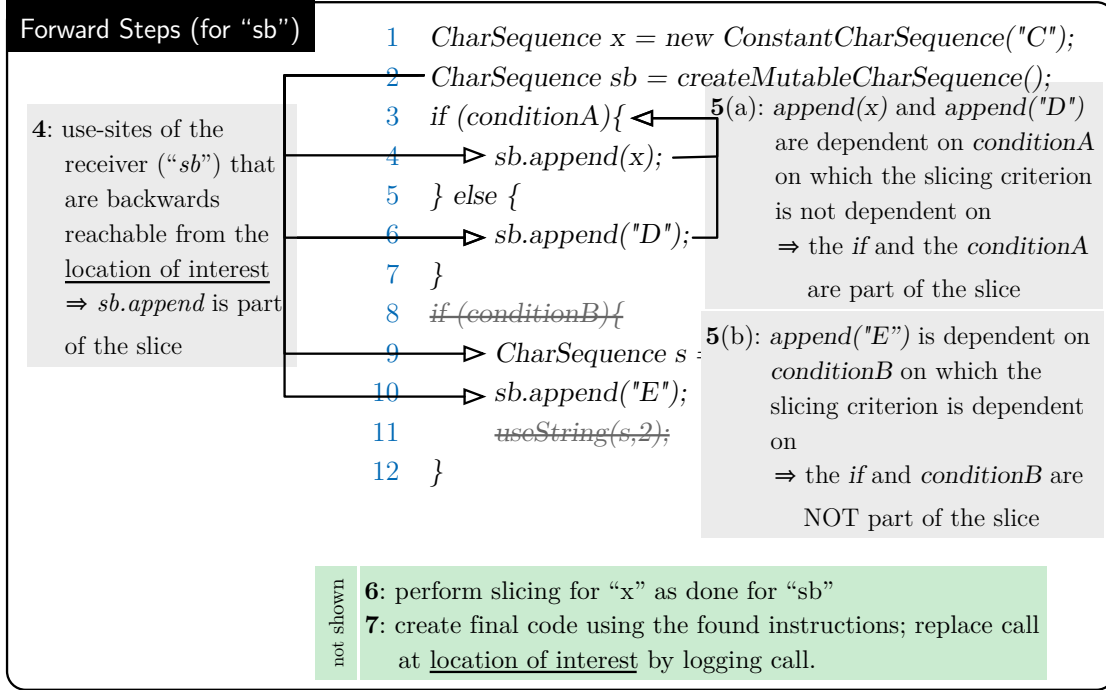


Figure 10.3.: Example of the slicing process for the parameter `s` of `useString` — Showing the necessary forwards steps

LoIs are pointed at by dashed arrows. Consequently, we establish in step 1 that the call in Line 11 is the *LoI*, and the call in Line 9 is the *s_{crit}*. In step 2, the algorithm determines the receiver object on which `toString()` is called, i.e., `sb`. Hence, in step 3, the algorithm adds the definition site of the receiver to the slice, which is the call in Line 2. The `if`-condition in Line 8 is not added to the slice because it would potentially prevent the execution of the slicing criterion (*s_{crit}*).

Having shown the steps 0 to 3, we continue our example in Figure 10.3 by showing the forward phase of the algorithm. Starting from Line 2, we perform a forward phase since no backward phase is necessary.

In step 4, the algorithm identifies the use-sites of `sb` and, since all of them are backward reachable from the *LoI*, it adds the Lines 4, 6, 9, and 10 to the slice. The algorithm conservatively adds Line 10 to the slice because it does not anticipate that after Line 9, nothing mutates the string.

In step 5(a), the algorithm processes the calls in Lines 4 and 6 as follows. The first call in Line 4 uses the variable `x` and, therefore, our algorithm adds the defining instruction in Line 1 to the slice. Additionally, it adds the `if`-instruction in Line 3 to the slice because both Line 4 and 6 are control dependent on the instruction, but not *s_{crit}*.

When the algorithm processes in step 5(b) the call in Line 10, it identifies that the `if`-condition in Line 8 would prevent the execution of *s_{crit}* and thus does not add the

condition to the slice. Step 6 is not explicitly shown because it repeats the steps 2 to 5 but starts with the variable `x` in Line 4.

To summarize the steps 0 to 6, the resulting slice is the entire code from Figure 10.3, except the Lines 8 and 11 which we crossed out in Figure 10.3. In contrast to our approach, traditional slicing algorithms would include the condition in Line 8 into the slice, which may prevent the execution of the relevant code.

Step 7 creates the executable code, including a method to retrieve the value that the *LoI* would have used. The following section explains this step in more detail.

10.2.3. Executing Sliced String Usages

In order to obtain the deobfuscated string that would flow into the *LoI*, our algorithm extends the slice with a method call that logs the string. This call effectively replaces the *LoI* by a call to the logging method that retrieves at this point the deobfuscated string. Nevertheless, if necessary, the algorithm adds a return statement to the slice to ensure that the signature of the sliced method can remain unchanged. Consequently, depending on the declared return type, either `null` or the numeric value 0 is returned.

Given the extended slice, the algorithm replaces the body of the original candidate method with the slice. By changing only the method body, the algorithm evades countermeasures that use the name of the declaring class as well as the name of the sliced method in their logic. For instance, the obfuscation schemes *Stack Calls* and *Key is the Signature of Stack Calls* (cf. chapter 8) use this information to prevent the unintentional execution of their deobfuscation logic.

Even though the algorithm does not change the names of the class and the sliced method, it has to change multiple other context information to execute the desired method. For instance, if the class is abstract, the algorithm has to change it to concrete. As a result, all abstract methods are made concrete by returning default values of the declared return type. Additionally, the algorithm generates a superclass that implements all methods transitively called by the sliced method, and the class of the method extends the superclass. The extension of the superclass includes the changing of the class static initializer and calls to `super`. With this step, the algorithm evades the countermeasures *Static Initializer*, and *Object Initializer* (cf. Chapter 8). It also increases the likelihood that the initialization of our class containing the sliced method does not abort with an exception. Recall that we have no means to determine appropriate parameter values and, therefore, the algorithm always uses default values if required.

Given the execution environment of the sliced classes and methods, the algorithm includes all classes from the original application, except the modified ones to ensure that the slice is self-contained. Furthermore, as a replacement of the original `android.jar`, the algorithm uses an artificial JAR with method stubs that have to return default values of their return type (e.g., `null` or 0). All these transformations with our slicing approach, enable *StringHound* to circumvent all obfuscation schemes discussed in Table 8.1. Even if an obfuscator uses reflection, the slice runs successfully as long as the targets are part of the execution environment. However, the algorithm cannot execute native methods since

most Android applications do not compile their native parts for the x86 architecture on which *StringHound* performs its the slicing.

Finally, the algorithm reflectively calls the sliced method in its execution environment using default values for the parameters. Thus, the sliced method calls our logging method that records the deobfuscated string. By using a separate execution environment for each sliced method, the algorithm can continue the execution of other sliced methods if one crashes or its runtime exceeds a specified time limit. This procedure ensures that no other slices are affected, and only the affected results will be missing. We chose to call the sliced methods with default values because they cause no overhead. Nevertheless, our approach does not depend on them and can also support more advanced methods for determining the parameter values such as fuzzing.

In order to demonstrate the creation of our execution environment, Listing 10.2 shows an abstract class that contains in its constructor an obfuscated string embedded into a call of the deobfuscation method. The deobfuscation method decodes the obfuscated string using the Base64 class from the Android class library. Additionally, the constructor of the code calls an abstract method that takes the deobfuscated string and another string as parameters.

After the slicing and preparation for execution, Listing 10.3 shows the resulting class. First, the algorithm slices the constructor by replacing the call to the abstract method with a call to our `SlicingLogger` and the removal of the assignment to the class field `o`. Since the deobfuscation code does not need the abstract method and the field, our algorithm removes them from the class. Additionally, because the JVM cannot call methods in abstract classes using reflection, the algorithm removes the `abstract` access flag of the class under analysis. Furthermore, it adds an extension to `OurSuperClass` to manipulate the execution of the constructor and execute the constructor reflectively using the empty string as a default value. If the algorithm needs more complex objects as default values, it transitively creates objects using constructors of the object classes that take no or a few parameters.

```

1  abstract class AbstractClass {
2
3      String o;
4
5      public AbstractClass(String s) {
6          this.o = doSomething(decrypt("QzovV2luZG93cy9TeXNOZW0zMj8="), s);
7      }
8
9      public String decrypt(String cipherText) {
10         try {
11             return new String(Base64.decode(cipherText, 0), "UTF-8");
12         } catch (UnsupportedEncodingException e) {
13             e.printStackTrace();
14             return null;
15         }
16     }
17
18     public abstract String doSomething(String f, String s);
19 }

```

Listing 10.2: Java code of string deobfuscation in an abstract class before slicing

```

1  class AbstractClass extends OurSuperClass {

```

10. String Deobfuscation

```
2
3     public AbstractClass(String s) {
4         SlicingLogger.log(decrypt("QzovV2luZG93cy9TeXNOZW0zMj8="));
5     }
6
7     public String decrypt(String cipherText) {
8         try {
9             return new String(Base64.decode(cipherText, 0), "UTF-8");
10        } catch (UnsupportedEncodingException e) {
11            e.printStackTrace();
12            return null;
13        }
14    }
15 }
```

Listing 10.3: Java code of string deobfuscation in an abstract class after slicing

Given *StringHound*'s classifiers and its slicing and execution procedure, we evaluate its performance in the next section.

10.3. Evaluation

For the evaluation of *StringHound*, we investigated the following research questions:

RQ1: How effective is *StringHound* compared to state-of-the-art deobfuscation tools?

RQ2: What is the prevalence of string obfuscation in the wild?

RQ3: Which classifier identifies most of the obfuscated strings?

RQ4: What kind of data is hidden using string obfuscation?

RQ5: Does string obfuscation hamper security analyses?

RQ6: What runtime performance does *StringHound* have?

To answer these questions, we performed three studies (a) comparing *StringHound* with other string deobfuscators, (b) assessing the performance, and (c) runtime of *StringHound* on real-world apps.

The setup consisted of a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The analyses were executed using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory and a 5s timeout for a single string deobfuscation.

For our studies, we used five different sets of APKs. The first set consists of the validation set of apps from the F-Droid used in Section 9.3. The second set consists of 100,000 APKs randomly selected from AndroidZoo [HAB⁺16]. The third data set consists of the Top 500 most common apps based on AndroidRank [And19]. The fourth set consists of apps that were available on the Play Store in 2018 and were classified as malicious by at least 10 AV vendors in VirusTotal [vir19]. Finally, the last set consists of 230 Android malware samples from Contagio [Con19b], containing current and past malware families.

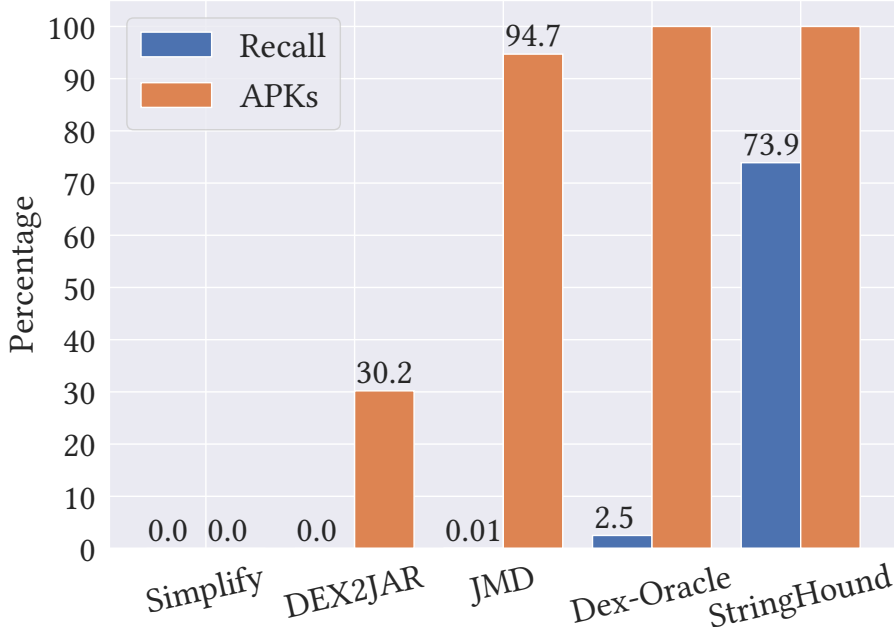


Figure 10.4.: Recall and successfully processed APKs

10.3.1. Comparison with Other Deobfuscators

We evaluated *StringHound* against *Dex-Oracle 1.0.5* [Dex20], *Simplify 1.2.1* [sim20], *JMD 1.61* [Jav20a], and *DEX2JAR 2.0* [Dex19b]. To the best of our knowledge, these are the only freely available tools suited to deobfuscate string values.

As input for the deobfuscators, we randomly picked 1,000 apps from the data set described in Section 9.3, which we have not previously used to train our classifiers. Two comparison metrics are used: (a) percentage of APKs processed without termination errors; and (b) recall, which we define as the percentage of *unique deobfuscated strings* over *all unique strings* in the original apps. We discard the precision metric since our data set contains only obfuscated strings. Therefore, no false positives could occur (i.e., plain strings identified as obfuscated). However, the false positives produced by the *String Classifier* and the *Method Classifier* restrict also *StringHound*'s false positive rate. The results are summarized in Figure 10.4. In the following, we discuss each deobfuscator individually.

As described in Section 10.1, *Simplify* [sim20] is limited for deobfuscation methods that do not depend on any state and use only constants. Unfortunately, *Simplify*'s re-engineered APKs were completely broken and could not be analyzed to produce results. Hence, Figure 10.4 reports 0% for both values. In order to get any viable result, we tried all possible configurations of *Simplify*, executed it on different operating systems, and tried to process different kinds of APKs. However, *Simplify* produced only broken APKs that contained no strings and were not executable.

DEX2JAR [Dex19b] needs the deobfuscation methods to be executed as user input.

10. String Deobfuscation

We applied our classifiers to each app and used its output as input for *DEX2JAR*. Providing the same deobfuscation methods to *DEX2JAR* and *StringHound* enables a fair comparison of the two. However, *DEX2JAR* processed only 30% of the APKs without errors, and it was unable to deobfuscate a single string, resulting in a 0% recall. *DEX2JAR*'s assumption that deobfuscation methods are in the same class as the obfuscated string caused these weak results. Moreover, *DEX2JAR* assumes that all values flow without any indirections into a provided deobfuscation method. However, the values can also be the result of other accesses or computations. Unfortunately, none of the obfuscation techniques from Chapter 8 matches *DEX2JAR*'s required conditions.

Since *JMD* [Jav20a] has many restrictions for the search of deobfuscation method signatures, the usage of only one constant string instructions, and the usage of constant keys, these restrictions lead to its poor performance. While *JMD* successfully processes 94% of the APKs, it only deobfuscates 0.01% of the strings.

As *JMD* and *DEX2JAR*, *Dex-Oracle* [Dex20] assumes that all values flow without any indirections into a deobfuscation method. As a result, it does not consider additional computations or accesses. Additionally, it supports only a fixed set of possible signatures of deobfuscation methods. As a result, Figure 10.4 shows that even though *Dex-Oracle* processed all APKs without errors, it recovered only 2.5% of all obfuscated strings. Its strict assumptions match only very few deobfuscation methods found in the wild, leading to its low recall. However, these matches could also be coincidences since the used obfuscator has various other method templates (cf. Chapter 8).

StringHound was able to process all APKs with a recall of 73.9%. A detailed analysis of the 26.1% missing cases showed that every obfuscation scheme listed in Table 8.1 occurred in the false-negative set. Furthermore, the analysis revealed that either the execution environment, surrounding the sliced method, is too complex to be modeled with our default values (cf. Section 10.2.3) or the classifiers were not able to identify the obfuscated strings (cf. Section 9.5.3). However, the high recall confirms the effectiveness of our approach, which does not suffer from the various limitations of the state-of-the-art deobfuscators.

Observation 26 *Unlike StringHound, other deobfuscators do not 'automatically' identify all analyzed obfuscated strings. As a consequence, other deobfuscators either use all methods of the app or get the deobfuscation methods as user input. Such a brute-force approach does not scale to large data sets. Given these results, StringHound is more effective than all other analyzed deobfuscators (RQ1).*

10.3.2. Findings in the Wild

In this section, we use *StringHound* to assess the prevalence of string obfuscation in the wild and the categories of obfuscation string usages. For this study, we use the 100,000 apps from AndroidZoo [HAB⁺16], the Top 500 most common apps based on AndroidRank [And19], the apps that were available on the Play Store in 2018 and were classified as malicious by at least 10 AV vendors in VirusTotal, and the 230 Android malware samples from Contagio [Con19b].

Prevalence of Obfuscated Strings in the Wild

We begin with a study to measure the prevalence of obfuscated strings in the wild. For the measurement, we apply *StringHound* to 100,000 apps from Section 8. In order to avoid false positives, we exclude all constant strings from our findings and count the remaining ones, which we refer to as newly revealed strings.

For the identification of *newly revealed strings*, we delete all constant strings in the analyzed app from the substrings of the deobfuscated string. In order to measure the rate of the newly revealed content, we use the length of the remaining string divided by the length of the original deobfuscated string. Removing shorter substrings hinders the removal of longer ones. For instance, given a deobfuscated string “Hello”, if “el” is removed then, we end up with “Hlo” and can no longer remove a string like “ello”. To avoid this effect, we first remove longer substrings before we remove the others by lexicographical order.

Given the measure of the newly revealed content per deobfuscated string, we calculated the number of APKs containing newly revealed strings. In our study in Chapter 8, we discovered that obfuscators may manipulate only parts of the strings in an app, and may also hide obfuscated strings in other data structures. These two findings lead to the following observation:

Observation 27 *StringHound invalidates the claims of previous studies [Don18, Mir18, WR17] that less than 5% of the apps contain obfuscated strings because we discovered that 76% of the 100,000 apps contain obfuscated strings (**RQ2**).*

Table 10.3.: Percentage of obfuscated strings identified by both classifiers

Classifier	Percentage
<i>String Classifier</i> (MC)	28%
<i>Method Classifier</i> (SC)	77%
Overlap (Both)	5%

In the scope of our investigations, we also measured the ratio of newly revealed strings per classifier. The results in Table 10.3 indicate that the *String Classifier* detected 28%, and the *Method Classifier* 77% of the newly revealed strings.

Observation 28 *These findings provide empirical evidence that both classifiers are needed since their findings only overlap with 5% of the newly revealed strings. The Method Classifier (MC) identifies most of them (**RQ3**). However, the String Classifier provides at least $(SC_{new} - Both_{new} =)$ 23% of newly revealed strings and, thus, it is also necessary for StringHound to achieve a higher total recall.*

Categorization of String Obfuscation

In order to understand the usage of string obfuscation in the wild, we used regular expressions to categorize all deobfuscated strings in different classes. Table 10.4 shows the

10. String Deobfuscation

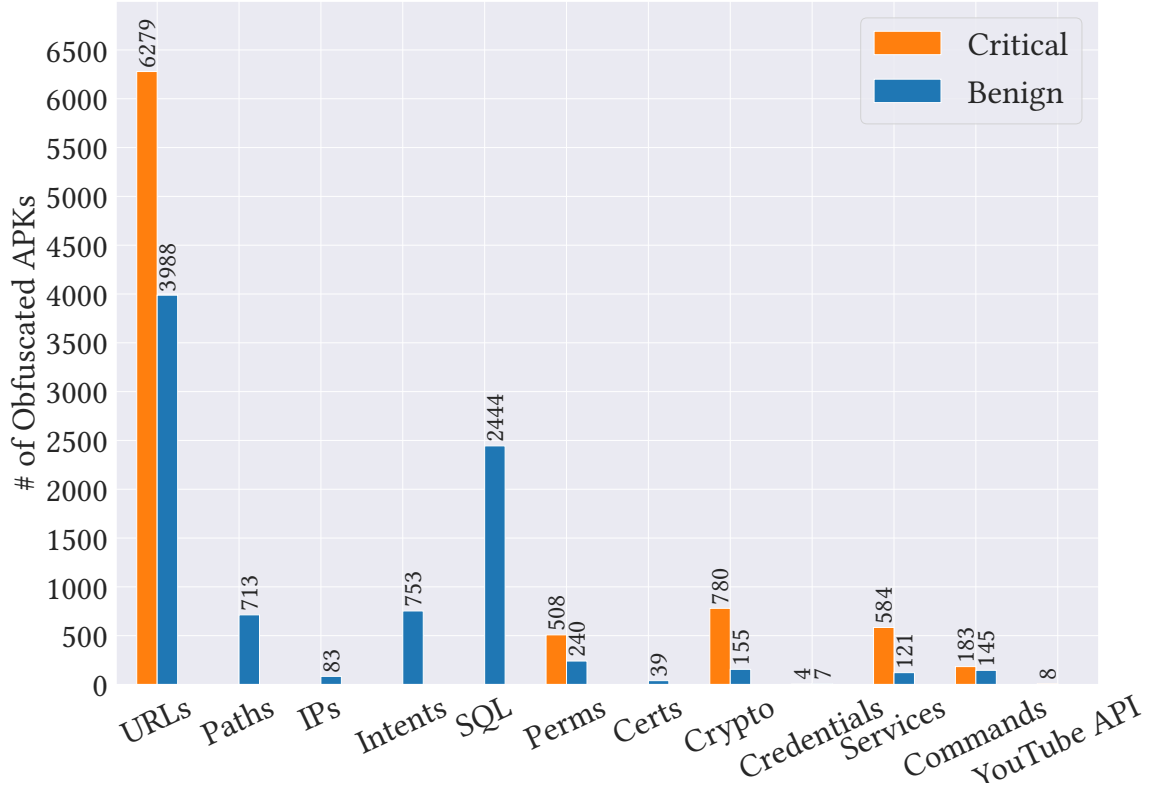


Figure 10.5.: Categories of string obfuscation in the 100k apps.

regular expressions we used to categorize deobfuscated strings. Our regular expressions for URLs are not limited to the typical HTTP(S) form but also matches any scheme, such as content URLs. Furthermore, we use a pattern for IPs to match non-URL related communications. The regex for paths describes absolute paths of the Android operating system matching not only directories and file paths but also absolute paths to executables. The patterns for intents and permissions match Android’s standard definitions. The regular expression for SQL statements matches strings with common keywords for querying and manipulating tables. Finally, we identify certificates by their Base64 encoded prefix of the first three characters.

Altogether, we defined 12 regular expressions for matching URLs, file system paths (Paths), IPs, intents, SQL statements (SQL), permissions (Perms), certificates (Certs), cryptography algorithms (crypto), credentials, system services (Services), commands, and API keys [MMR19]. We applied them to all deobfuscated strings in our data sets (discarding apps without newly revealed strings) and counted their matches to quantify the prevalence of each kind of usage.

Figure 10.5 shows a categorization of the resulting deobfuscated strings in the 100,000 apps. We divide the bar chart into critical and benign apps with obfuscated strings. Many of the critical strings we identified by counting the following facts. First, we

Table 10.4.: Regular expressions for the evaluation of deobfuscated strings in the wild.

Name	Regex
URL	<code>\w+://[^\"]+.*</code>
Paths	<code>\/\w+[\./]+</code>
IP	<code>.*\b([0-9]{1,3}\.){3}[0-9]{1,3}\b.*</code>
Intents	<code>android.intent\..*</code>
SQL	<code>.*(select.*from update.*set insert into delete from create table drop table truncate table).*</code>
Permissions	<code>android.permission\..*</code>
Certificates	<code>MI.+</code>
Cryptography algorithms	<code>MD2 MD5 SHA\.-?1 ECB DES</code>
Credentials	<code>.*user.*\..*passw.*</code>
System Services	List of System Services with wild cards (e.g. <code>.*power.*</code>)
Commands	List of Commands with wild cards (e.g. <code>.*su.*</code>)
Youtube API Key [MMR19]	<code>AIza[0-9A-Za-z\-_]{35}</code>

identified more HTTP requests than HTTPS, which may lead to security issues [Pro19]. Second, developers request permissions but are not aware that these permissions are also used by ad libraries to access private data via obfuscated strings. Third, some apps use insecure cryptography algorithms such as DES, AES with ECB mode, or MD2 via obfuscated strings. Fourth, some apps send credentials hidden in obfuscated strings using the HTTP GET method to login to their services. Fifth, some apps provide dangerous accesses (e.g., the location of the device) via services that are requested using obfuscated strings. Sixth, rooted phones execute commands, hidden in obfuscated strings, to grant root access. Last, YouTube API keys, hidden in obfuscated strings, can be used to consume the developer’s API quotas.

Observation 29 *Using StringHound on the 100,000 apps, we identified in the obfuscated strings critical usages of URLs, piggy-backed permissions, insecure cryptography algorithms, hard-coded credentials, dangerous services, root commands, and API keys (RQ4).*

Due to this study, we identified many vulnerabilities that hamper the security of the app user or developer (RQ5).

Context Analysis of the Categories

While the 100,000 apps contain a large variety of statistical findings, we have no insights into apps that belong to the extreme fields of the Android ecosystem. To get more insights about such apps, we chose three different data sets to get an understanding of these kinds of fields and the context of *StringHound*’s findings. These data sets consist of the top 500 most installed apps in the Play store, and two malware sets to analyze current (malware 2018) and past (Contagio) obfuscated malware. Figure 10.6 shows a categorization of the deobfuscated strings. Each bar corresponds to the percentage of

10. String Deobfuscation

APKs from a data set containing at least one deobfuscated string in the given category. Each category comprises a group of three bars, where each corresponds to one data set.

The first bar shows that 60% of the Contagio malware obfuscates strings, mostly paths (40%), URLs (12%), or intents (5%). A detailed analysis revealed absolute paths of commands trying to open a command shell or of further APK or DEX files hidden in the resources of the app containing the malware’s actual payload. One path establishes a connection to a Command & Control server (AnserverBot [GZZ⁺12]). Furthermore, we found paths to files on the SD card and to DHCP settings, which are exploited by the DroidKungFu2 malware [KCS16]. Our regex for URLs matched locations of browser settings that attackers may use to build a profile of the underlying mobile phone. Additionally, we found URLs whose sites provide the geolocation of the accessing IP address. We also found URLs to ad networks that may profile the user’s phone. The regular expression for intents matched an action that resets the default page of the browser to either show pages of ad networks or to track the user’s behavior. Furthermore, the intent regex discovered an action, which queries the cell phone number to reveal the identity of the user. Finally, we also found an action that performs phone calls.

In the malware set from 2018, 35% of APKs use string obfuscation to hide various interactions with the Android operating system. We matched URLs that lead to ad networks, which can track the user’s interactions and build profiles of users. Additionally, some URLs access the user’s calendar and can reveal detailed information about their schedules. The tracking of interactions, in combination with profiling, violates the user’s privacy. We also found hidden paths of an APK holding its malicious payload. Moreover, we revealed paths to operating-system commands that access hardware and sensor data to profile a phone. Findings regarding intents and permissions indicate that malware uses intents to call or send SMS to premium numbers. Additionally, the malware tries to locate or profile a user by accessing personal calendars, accounts, or states of a phone. Compared to the Contagio malware-set, more recent malware focuses on leakages of private data, causing financial damage to the unknowing user.

Observation 30 *Current malware in the Play store makes less use of string obfuscation (35% compared to 60%) and focuses more on hiding leakages of private data. Without StringHound, one would miss information that is essential to detect remote command execution, even causing financial damage to the user, and leakages of private data in at least 35% of recent malware.*

Surprisingly, string obfuscation is more frequently used in the Top 500 apps than in the 100,000-set of apps (89% vs. 76%), even more frequently than in malware. Our evaluation shows that 33% of the apps use obfuscated URLs. Some of those URLs track user’s IDs and IP through an ad network.

These actions directly violate users’ privacy. A detailed review of the findings showed that all ad libraries contain obfuscated URLs and paths. We also analyzed how many apps use string obfuscation only in ad- and third-party libraries¹. This analysis revealed

¹To this end, we filtered our findings by the list of ad-library package names from Chapter 8 and by a list of common libraries [LKLT⁺16].

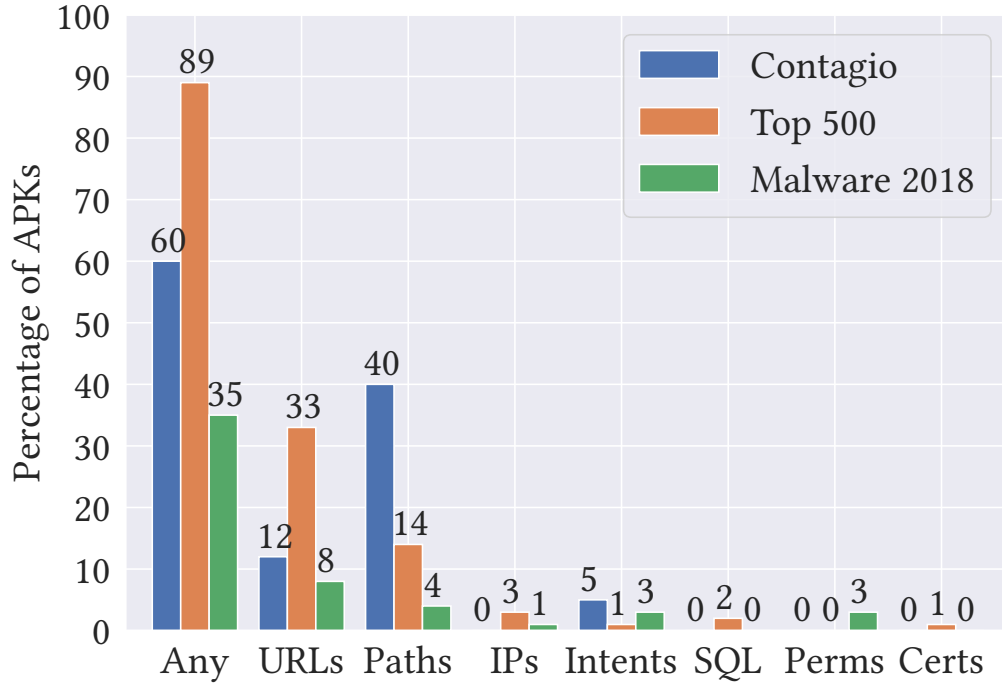


Figure 10.6.: Categories of string obfuscation in the wild.

that ad libraries contain 63.52%, other libraries contain 10.64%, and the remaining 25.84% of all obfuscated strings in the Top 500 data set are in the app itself.

Observation 31 *We found that all sorts of apps frequently contain string obfuscation. Ad libraries are responsible for over 63% of these strings that hide the tracking of users. This result is alarming since neither the user nor the developer of the app is aware of the added functionality. With StringHound the developer could check the content of the used ad library and choose an appropriate alternative.*

During our analyses, we found two games for children that contained obfuscated privacy violations in the Top 500 data set. We manually analyzed their code and found that they collect and transmit sensitive information on the user’s mobile phone. The leaked information includes build, connectivity, debug, runtime, telephony, Android version, and hardware data, which may build a user profile. Code related to this data collection functionality resides in a stealthy package mixed into the integrated Android support library. The app additionally checks for the `SuperUser.apk` that grants root access to the mobile phone. We reported both apps to the app store since, according to AndroidRank [And19], at least 20 million devices installed these suspicious apps.

Additionally, we uploaded both apps to VirusTotal [vir19] to test them and made the following observation:

Observation 32 *Virus scanners do not flag suspicious privacy violations, since the scanners of VirusTotal showed no findings, besides the usage of dangerous permissions. StringHound equips an analyst to search for more kinds of violations than provided by virus scanners.*

10.3.3. Runtime Performance

In order to evaluate the runtime performance of *StringHound*, we measured the average runtime per APK and per slice by running it on the Top 500, and the two malware data sets from Section 10.3.2. While the first measure shows the runtime of our approach for different APK sizes, the second one can be used to approximate the analysis time for a given APK. All performance measures indicate that *StringHound* is fast and ready for practical use.

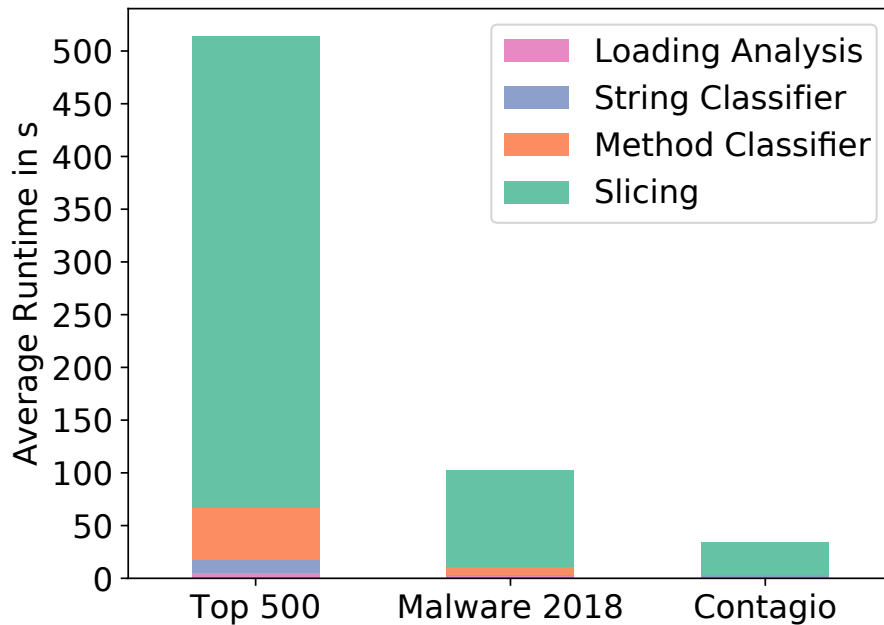


Figure 10.7.: Average runtime for the top 500 and the two malware data sets

Figure 10.7 shows the average runtime per APK. Each bar corresponds to one data set, and we split it into the time needed (a) for analysis loading, (b) *String Classifier* execution, (c) *Method Classifier* execution, and (d) the building and execution of slices.

With the results in Figure 10.7, we show the runtime for different APK sizes. As shown, the processing of the Top 500 data set needs, on average, up to 20 times more time per APK than processing the Contagio data set. The reason for this high discrepancy is a large amount of library code in the APKs of the Top 500 data set. As mentioned in Section 10.3.2, 74.16% of the obfuscated strings are found in libraries (i.e., ad + other libraries), and these are up to 14 times larger in code size than APKs from the

Contagio data set. We can reduce the time it takes to analyze such apps by using tools that separately analyze the library code and reuse these analysis results. Another observation is that across all three data sets, slicing consumes most of the execution time. Hence, improving the performance of the slicing would speed up the entire analysis.

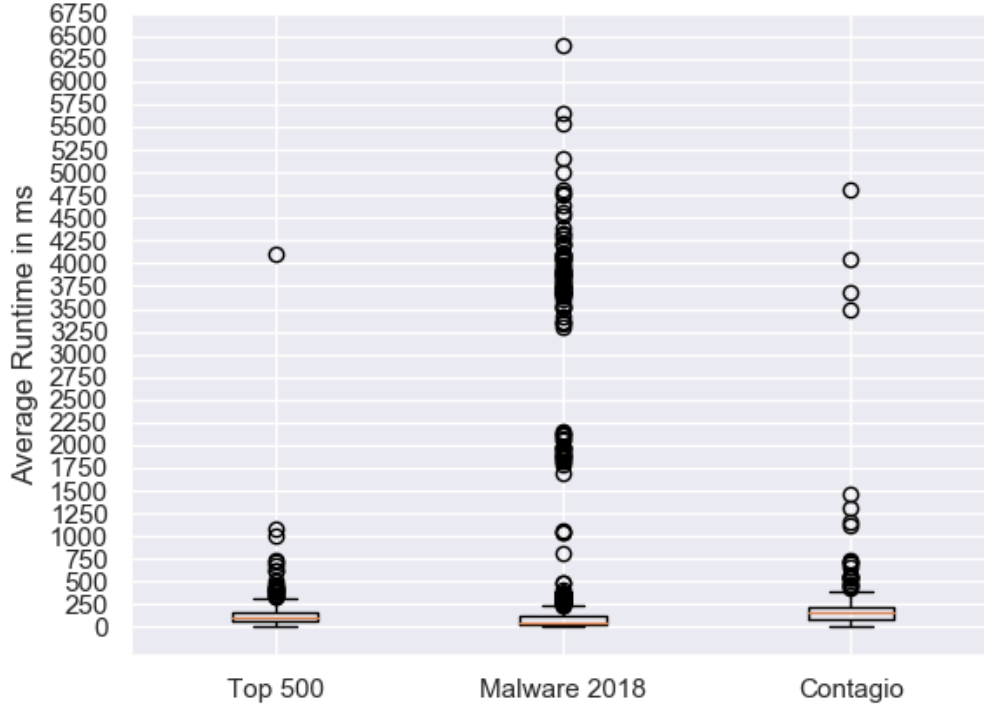


Figure 10.8.: Runtime per slice for the top 500 and the two malware data sets

For the calculation of the average runtime per slice, we measured the mean, median, and also the 95%-quantile for each slice of all three data sets. While Figure 10.8 shows that all these measures are below 250 ms, we have many outliers that are not only caused by the different code sizes but also by the complexity of the code.

Observation 33 *From the values of the median and quantiles, we conclude that building and executing a single slice takes on average less than 250 ms (RQ6).*

Given the observation that slicing consumes most of the execution time and the execution of a single slice takes less than 250 ms, the only improvement to speed up the performance is to parallelize the building and execution of single slices.

10.4. Threats to Validity

In this section, we discuss threats to the validity of *StringHound* that can be subject to further consideration in future work.

Driven by the study of obfuscation schemes, *StringHound* uses intra-procedural slicing and default values for parameters to recover automatically obfuscated strings. As a result, the slice's execution may fail if it expects values, which differ from our injected defaults. However, we can address this limitation by fuzzing the expected values. Given a field or parameter, fuzzing guesses their values by their data-dependencies or using symbolic execution to discover possible value ranges. Additionally, If a decryption key is not present in the app code, we cannot deobfuscate strings that obfuscators encrypted using this key. For instance, if the app code needs to download a deobfuscation key over a network, and the download is only executable under particular circumstances.

Furthermore, obfuscators can use fields and parameters to perform inter-procedural obfuscation. However, to perform it automatically, they need to identify the call order of the fields and parameters. This call-order is not readily identifiable because of the limitations of current call graph analyses for Android. Of course, making *StringHound* inter-procedural is an obvious alternative, but coping with potential inter-procedural obfuscation schemes is a trade-off between soundness and performance.

Further threats for *StringHound* are the dynamic usage of external packages or resources, encrypted classes, and native code. While these techniques would evade our approach, the combination of other tools [KKR15] with *StringHound* can mitigate these threats. However, obfuscators may use developer certificates [ZLQ⁺18] or other external resources from the app to prevent the execution of the extracted code. Currently, no deobfuscator can handle such techniques.

10.5. Conclusion

This chapter shows how and why string obfuscation is used in real-world Android and Java apps. We presented *StringHound*, our approach to recover obfuscated strings. *StringHound* significantly improves over state-of-the-art deobfuscation tools. We also presented a large-scale study on the use of string obfuscation in benign and malicious apps, revealing highly relevant findings.

We provide empirical evidence that string obfuscation is commonly used across malware, and 100,000 apps from Google's Play Store. This evidence invalidates statements by previous research, suggesting that string obfuscation is rarely used in practice. By undoing string obfuscation, we revealed abundant problematic string usages in the wild: Critical internet accesses, piggy-backed permissions, insecure usage of cryptography algorithms, hard-coded passwords, and available YouTube API keys. We have found malware concealing hidden commands and communication endpoints and spyware-like behavior in two apps in the Top 500 set. Our studies have shown that libraries account for a significant amount of obfuscated strings in benign apps. Many findings in the ad libraries reveal serious privacy issues.

Summary

The presence of obfuscated strings impedes the analysis of apps, e.g., to check their compliance with privacy regulations or to inspect them for detecting malware [RAMB16, MW17]. String obfuscation can hide paths, URLs, and intents that can be used to track the activities of a user or open shells on the user’s device to activate malicious payload remotely. To address the issues with string obfuscation, we conducted a study of string obfuscation techniques in the wild. Using the results of the study, we developed two detectors to identify obfuscated strings, and our deobfuscator *StringHound* that uses both detectors to reveal the content of obfuscated strings.

In Chapter 8, we analyzed 640 unique ad libraries extracted from 100,000 randomly selected apps of the AndroZoo [HAB⁺16]. We examined the ad libraries for techniques that obfuscate strings. According to previous works [SGC⁺12, RNVR⁺18, DMY⁺16, SKS16, CFL⁺17], many of the ad libraries contain such strings, and therefore, we use the libraries for the identification of string obfuscation techniques. To identify obfuscated strings, we analyzed all constant strings for abnormal usage of non-alpha-numeric characters. Additionally, we analyzed all code locations that get non-string arguments but return strings to identify deobfuscation code.

Using this procedure, we discovered 21 unique string obfuscation techniques and analyzed their functionalities for further experiments. In particular, we focused on mechanisms used to protect obfuscated strings from unauthorized deobfuscation. Our study showed that all identified string obfuscation techniques place their deobfuscation logic in the same class as the obfuscated string, making its logic easy to target.

We also discussed the different mechanisms that protect obfuscated strings and the shortcomings of their current implementations against automatic analyses. Furthermore, we argued that fully-automated string obfuscation is challenging without additional input from the obfuscator’s user.

In Chapter 9, we introduced two approaches based on the analysis from Chapter 8. These approaches complement each other to identify obfuscated strings in apps. The first approach *String Classifier* directly analyzes strings with a classifier that identifies obfuscated strings based on their characteristics. Furthermore, the second approach *Method Classifier* compares methods with ones containing known deobfuscation logic to identify obfuscated strings hidden in other data structures.

String Classifier analyzes strings for words from different natural languages and uses different features from cryptanalysis, known formats, and word characteristics to identify obfuscated strings. Whereas, the *Method Classifier* extracts a specific representation of each method in an app and compares the instructions of this representation with known deobfuscation logic using the correlation between the numbers of different instructions.

Since most other tools cannot detect obfuscated strings on an individual basis, we

10. String Deobfuscation

evaluated the *String Classifier*'s effectiveness only on our data set without comparing it with other tools. However, with a F_1 -score above 94%, the *String Classifier* is well suited to identify obfuscated strings in the wild.

While we had for the *String Classifier* a well-defined data set and no comparison tools, we had for the evaluation of the *Method Classifier* tools for the comparison, but no well-defined data set. The collection of a well-defined data set was impossible because many obfuscators do not document the methods which they use for deobfuscation. Consequently, we used a technique from graphic recognition that assesses multiple tools without the necessity of a ground truth. After we established the necessary conditions for the usage of this technique, we could analyze the relations between each tool's precision and recall. The results indicate that the *Method Classifier* identifies most of the deobfuscation logic.

Finally, Chapter 10 shows how and why string obfuscation is used in real-world Android apps. We presented *StringHound*, our approach to recover obfuscated strings. *StringHound* significantly improves over state-of-the-art deobfuscation tools. For the deobfuscation, *StringHound* uses *String Classifier* and *Method Classifier* to detect obfuscated strings. Afterward, it extracts all necessary values and logic to trigger the string's deobfuscation and executes it. For the execution, *StringHound* integrates the logic in an environment that is sliced from the original one to evade techniques that try to protect the obfuscated string from unauthorized deobfuscation.

In our large-scale study on the use of string obfuscation in benign and malicious apps, we also revealed highly relevant findings for the security and privacy of app users.

We provide empirical evidence that string obfuscation is commonly used across malware and 100,000 apps from Google's Play store. This evidence invalidates statements by previous research, suggesting that string obfuscation is rarely used in practice. By undoing string obfuscation, we revealed abundant problematic string usages in the wild: Critical internet accesses, piggy-backed permissions, insecure usage of cryptography algorithms, hard-coded passwords, and available YouTube API keys. We have found malware concealing hidden commands and communication endpoints and spyware-like behavior in two apps in the Top 500 set. Our studies have shown that libraries account for a significant amount of obfuscated strings in benign apps. Many findings in the ad libraries reveal serious privacy issues.

Part IV.

Conclusion and Outlook

11. Conclusion

In this chapter, we present an overview of this dissertation’s findings. We start reviewing the contributions of this dissertation and close with a discussion about open challenges for future research.

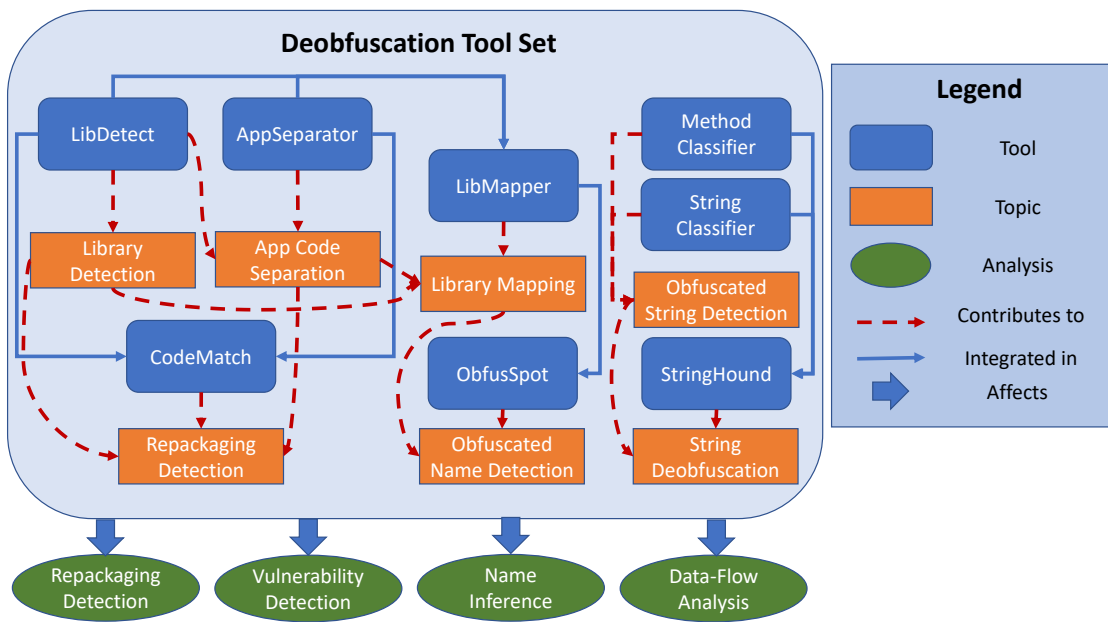


Figure 11.1.: Overview of contributions

Figure 11.1 shows the topics to which we contributed our tools and the analysis directions that are affected by our contributions. The topics of library detection and app code separation contribute to the ones of repackaging detection and library mapping. For this reason there are connections from the two tools *LibDetect* and *AppSeparator* to the others *CodeMatch* and *LibMapper*. The topic of library mapping contributes to the one of obfuscated name detection. Finally, the topic of obfuscated string detection is performed by two tools used as pre-analyses for the deobfuscation of these strings.

In the following section, we summarize our contributions to each topic. In the closing discussion, we explain how our contributions affect the analyses depicted in Figure 11.1.

11.1. Summary of Results

In this dissertation, we contribute to the fields of obfuscation detection and deobfuscation. Analysts can discover the library code that is mixed with app code by using our library detection tool *LibDetect*. If they are not interested in the specific libraries, they can filter out library code with *AppSeparator*. Both tools can be used to filter library code for repackaging detection with *CodeMatch*.

Furthermore, our tool *LibMapper* supports security analysts in understanding the internal structures of obfuscated library methods that might be used for malicious purposes. Since we only deobfuscate library code, an analyst might use a name inference approach to understand obfuscated app code. However, it is hard to find appropriate code samples to train such a name inference approach because many of the samples contain obfuscated names. Moreover, analysts do not have suitable tools to identify obfuscated names in those code samples. For this purpose, we developed *ObfusSpot* that not only identifies that an app contains obfuscated names but can also pinpoint the exact location of an obfuscated name.

Besides obfuscated names and libraries, analysts scan the app’s data for potential vulnerabilities or known signatures of malware. However, obfuscated strings hinder such analyses. As a consequence, we developed *String Classifier* and *Method Classifier* which complement each other in the detection of obfuscated strings. Furthermore, we proposed a string deobfuscation tool called *StringHound*. While *String Classifier* identifies obfuscated strings by anomalous characteristics, *Method Classifier* examines logic used to deobfuscate these strings.

11.1.1. Library Detection

For the detection of obfuscated libraries, we developed *LibDetect* that uses five representations for its detection algorithm. Each representation abstracts additional information from the code to identify more obfuscated methods without losing precision. After matching methods, *LibDetect* aggregates the methods to identify the classes of known library versions.

For the evaluation of *LibDetect*, we compared its precision, recall, and F_2 -measure with an approach that uses a white list of *Common Libraries* [LKL⁺16] and a library detection approach that is more advanced than a white list called *LibRadar* [MWGC16]. To compare the approaches, we downloaded 1,000 apps from five different app stores and manually identified all library classes. The comparison showed that *LibDetect* identifies more than six times more library classes, and its F_2 -measure is five times higher than the ones of the best tool *LibRadar*. However, in contrast to the other tools, *LibDetect* focuses on a high recall. Consequently, its precision is up to 24% smaller than the values of the other tools.

11.1.2. App Code Separation

Since many approaches need to separate app code without detecting all individual libraries in the code, we developed *AppSeparator* that separates app code without the need for a database of all library method representations. The design of *AppSeparator* is based on the separate compilation assumption [AL12], which states that libraries were already compiled before their integration into apps; therefore, libraries have no dependency on any specific app. Using this assumption, *AppSeparator* performs the following steps to separate app code.

First, it extracts all classes containing entry points and all classes from the main package of an app. With these classes, *AppSeparator* transitively identifies all additional classes that access the ones identified earlier.

Second, obfuscation can blur the lines between app and library code by inserting into the library code false references to app classes. To mitigate this effect, *AppSeparator* calculates the fan-in and fan-out values of all classes to train a classifier that decides whether some of the classes only belong to the main app code.

Last, *AppSeparator* uses this classifier to get a binary decision whether a class belongs to the app code or the library code.

For the evaluation of *AppSeparator*, we used the 1,000 apps from the evaluation of *LibDetect* and split them into 80% training set and 20% validation set. We used the former set to train our classifier and the latter to compare *AppSeparator* with *LibDetect* and *LibRadar*. The results suggest that *AppSeparator* is at least 11.55% more accurate in separating app code than the other approaches. Comparing our tool with and without classifier showed that the use of the classifier is responsible for the better results of *AppSeparator*, which has a 42.19% higher F_1 -measure than without the classifier.

11.1.3. Repackaging Detection

For the detection of repackaged apps, we developed *CodeMatch* that filters out library code and compares the remaining code with the code of original apps. If one app has a high similarity to an original one and is not produced by the same developer, it is considered repackaged. Before comparing the apps, *CodeMatch* also filters out small apps and apps generated by App Makers [app17a] because these apps are very similar without being repackaged.

For filtering library code, *CodeMatch* uses, depending on the code size of an app, either library detection or app code separation to reduce the library code's influence on the similarity measurement of the approach. *CodeMatch* performs the filtering using the integrated tools *LibDetect* and *AppSeparator*. While *LibDetect* has a large runtime, *AppSeparator* is not suited for small apps. After filtering, *CodeMatch* evades obfuscation techniques using the most abstract representation from *LibDetect*.

We evaluated *CodeMatch* by manually analyzing 1,000 app pairs based on their appearance and their code and comparing the results of *CodeMatch* with *FSquaDra* [ZGC⁺14], *ViewDroid* [ZHZ⁺14], *DroidMOSS* [ZZJN12], and *Centroid* [CLZ14] on this data set. The results show that *CodeMatch* outperformed all other approaches at least by 3.58%

11. Conclusion

because it handles large and small repackaged apps. Finally, we discovered that, on average, 15% of all analyzed markets contained repackaged apps.

11.1.4. Library Mapping

For the library mapping, we presented our approach *LibMapper* that maps the original names of classes, fields, and methods in libraries to their obfuscated counterparts. It uses *AppSeparator* to identify all library classes in an app and *LibDetect* to identify possible name candidates for library classes. Afterward, *LibMapper* determines the matching field and method names and combines each mapping for each of the suggested class candidates. Finally, *LibMapper* assigns the name mapping that has the highest probability to be the original counterpart of the obfuscated class.

For the evaluation of *LibMapper*, we compared its precision, recall, and F_1 -measure with the ones of *DeGuard* [BRTV16], the currently state-of-the-art name deobfuscation approach. To compare both approaches, we downloaded non-obfuscated apps from the F-Droid store [F-D19] and obfuscated them with different name-obfuscation configurations of *ProGuard* [Pro17]. The results show that *LibMapper* recovers up to seven times more obfuscated names than *DeGuard*.

11.1.5. Obfuscated Name Detection

Since many apps contain obfuscated names, it is tedious to find non-obfuscated and up-to-date libraries to supply a database for library mapping. As a consequence, we developed *ObfusSpot* that identifies obfuscated names in three steps. First, it performs an anomaly detection of names that do not conform to learned patterns of non-obfuscated names. Second, it measures each name’s occurrence frequency to identify names that have a higher frequency than non-obfuscated names because these are mostly unique. Finally, *ObfusSpot* uses *LibMapper* to identify the naming patterns of obfuscated library names and generalize these patterns to identify obfuscated names that were not found with the previous steps.

For the evaluation of *ObfusSpot*, we downloaded mapping files from GitHub repositories to train and test its classifiers. Obfuscators generate these files during the manipulation of names to enable developers to map names back to their origin, and some developers publish their mapping files to GitHub [git20]. We evaluated whether *ObfusSpot* generalizes on the app codebase allowed to identify more obfuscated names than the ones found by *LibMapper*. All results suggest that *ObfusSpot* is an accurate detector of obfuscated names and identifies at least 18% more obfuscated names than other approaches because they identify only names that are obfuscated with standard naming schemes. In the end, we analyzed the percentage of libraries that are only available in obfuscated form by analyzing 100,000 randomly selected apps from AndroZoo [ABKLT16]. Our analysis discovered that, in particular, ad libraries are mostly released in obfuscated form.

11.1.6. Obfuscated String Detection

To detect obfuscated strings, we first analyzed ad libraries to examine techniques used to obfuscate strings. According to previous work [SGC⁺12, RNVR⁺18, DMY⁺16, SKS16, CFL⁺17], many ad libraries contain such strings. To identify obfuscated strings, we analyzed all constant strings for abnormal usage of non-alpha-numeric characters. Additionally, we analyzed all code locations that get string or non-string arguments but return strings to identify deobfuscation code.

Using this procedure, we discovered multiple string obfuscation techniques and focused on mechanisms used to protect obfuscated strings from unauthorized deobfuscation. Our study showed that all identified string obfuscation techniques place their deobfuscation logic in the same class as the obfuscated string, making this logic easy to target.

We developed two approaches to target obfuscated strings that complement each other. The first approach, *String Classifier*, directly analyzes strings with a classifier that identifies obfuscated strings based on their characteristics. *String Classifier* analyzes strings for words from different natural languages and uses different features from cryptanalysis, known formats, and word characteristics to identify obfuscated strings.

We evaluated *String Classifier*'s effectiveness on an app data set that we obfuscated using the identified techniques from our study. The results indicate that *String Classifier* is well suited to identify obfuscated strings in the wild.

The second approach, *Method Classifier*, compares the instructions of methods with ones from known deobfuscation logic to identify obfuscated strings hidden in other data structures. *Method Classifier* extracts the *Structure Preserving Representation* (cf. Section 3.2.1) of each method in an app and compares this representation to the one of known deobfuscation logic by using the correlation of instructions between both representations.

For the evaluation of *Method Classifier*, we used a technique from graphic recognition to assesses multiple tools without the necessity of a ground truth. After we established the necessary conditions for the usage of this technique, we could analyze the relations between each tool's precision and recall. The results indicate that *Method Classifier* identifies at least twice as many deobfuscation methods as the other tools.

11.1.7. String Deobfuscation

Since obfuscated strings may hinder the analysis of data privacy concerns, we recover obfuscated strings using our tool *StringHound*. *StringHound* uses *String Classifier* and *Method Classifier* to detect obfuscated strings. Afterward, it extracts all necessary values and logic to trigger the string's deobfuscation and executes it. For the execution, *StringHound* integrates the logic in an environment that is sliced from the original one to evade techniques that protect the obfuscated string from unauthorized deobfuscation.

Our evaluation shows that *StringHound* outperforms the other deobfuscators by orders of magnitude. Additionally, it provides empirical evidence that string obfuscation is commonly used across malware and many apps from Google's Play store. This evidence invalidates statements by previous research [Don18, Mir18], suggesting that string obfuscation is rarely used in practice. By undoing string obfuscation, we revealed abundant

11. Conclusion

problematic string usages in the wild. Our studies further show that libraries account for a significant amount of obfuscated strings in benign apps. Many findings in ad libraries reveal serious privacy issues.

11.2. Closing Discussion

While obfuscation has some advantages such as reduction of the disc space of apps, it still poses a significant threat for app analysts since it renders the identification of private data leaks or security vulnerabilities harder.

To support analysts in their tasks, we contributed in this dissertation our tool set for the automatic detection and recovery of obfuscated apps. The tool set can assist in detecting obfuscated repackaged apps, identifying the use of obfuscated vulnerable libraries, or recover obfuscated strings for data-flow analyses. Additionally, using *ObfusSpot*, it supports several approaches in finding non-obfuscated code for their analyses. For instance, analysts can use *CodeMatch* to detect repackaged apps. Furthermore, they can use *StringHound* to identify vulnerabilities in obfuscated strings. Moreover, as we showed, *StringHound* can identify the disclosure of the geo-location by checking piggy-backed permissions. Additionally, with *LibDetect* or *LibMapper*, they can detect vulnerable libraries on different granularities. Both tools can reduce the effort for data-flow analysis by identifying all flows in a specific library and reusing the results in all apps that use the library.

In addition to the developed tool set, we analyzed multiple data sets in this dissertation. Using our analyses, we revealed the significant prevalence of obfuscated strings and the fact that about 4% of the libraries are released only in obfuscated form. We published these data sets to support future research. Although this dissertation addressed many of the threats caused by obfuscation, future research still faces several challenges that we have identified during our studies.

The first challenge is the rising number of apps that need to be analyzed. The Google Play store alone in 2020 already contains about 3 million apps [num20], and there are many more app markets in the world. As we mentioned in Section 10.3.3, tools need to process apps as fast as possible to handle these large numbers of apps.

Second, many approaches need to maintain a database to process current apps. As we mentioned in Section 3.4 and Section 6.5, keeping these databases up-to-date is a very challenging task since there exists no central register that contains all needed artifacts.

Third, in order to work properly, many approaches need an unchanged structure of an app. For instance, our tools need this condition to use the structure-preserving representation. However, current obfuscation techniques could change the code structure by flattening all methods' control flow.

Fourth, if an obfuscator encrypts a class, virtualizes the code, or performs another technique to hide the actual code of an app, many analysis approaches become useless. For instance, as mentioned in Section 9.6, if an obfuscator hides all obfuscated strings in an encrypted class, then neither the *String Classifier* nor the *Method Classifier* can detect these strings.

Finally, as mentioned in Section 10.4, obfuscators could improve their techniques to evade current analysis approaches. For instance, an obfuscator could use the developer certificate as a decryption key to protect an obfuscated string [ZLQ⁺18]. Since most analysis approaches do not extract the developer key during the deobfuscation process, they cannot reveal such strings. Another way to improve the techniques of obfuscators is to manipulate code and data until all analysis approaches can no longer be performed [XQE16].

12. Future Work

In this chapter, we point out future challenges that we consider particularly interesting and give suggestions on how these challenges could be addressed. Additionally, we provide an outlook of current obfuscation techniques that drive the described challenges.

12.1. Increasing Processing Efficiency

A future direction to process more apps is to split a data set into multiple subsets and start many instances of an analysis to process these subsets. However, some approaches process internal representations sequentially instead of parallelizing their processing. Approaches need to meet the tradeoff between both parallelization strategies to speed up data processing. To meet this tradeoff, we suggest as future direction an analysis platform that measures the parallelization opportunities of an approach and the hardware that should be used for the execution of an approach. With such a platform, an analyst could optimize either the internal structures for multithreading or the approach and its analysis data for multiprocessing.

Furthermore, since many analyses have an extended processing time, another future direction could be the development of multiple preprocessing filters with a short runtime, almost perfect recall, and precision as good as possible [WGMC15]. Such filters do not need to be separate approaches but could also be a simple categorization of continuous values in a database to improve the query speed for such values. For instance, *LibDetect* checks whether the matched class contains at least half of the number of instructions as the class under analysis. It could also use the number of instructions as a prefilter to consider only classes that meet this requirement by categorizing all classes in the database by their instruction number. A possible categorization for this count could be analogous to the collision rate of the method representations in Section 6.2.

Another further direction could be the automatic identification of possible candidates for such filters. For instance, if we have multiple approaches that can be used as filters, we can use machine learning to identify the best combination of filters based on the performance values of these approaches.

12.2. Keeping Representations Up-to-Date

We identified two sub-challenges to keep representations for library detection, library mapping, and repackaging detection up-to-date. The first one is to find an representation that can identify new artifacts even when they are obfuscated, and the second one is to support analysts with tools for their manual analysis.

Integrating New Obfuscated Artifacts To integrate a new artifact into the database of a tool, analysts need to identify and capture all associated elements of this artifact. For instance, if a library is not obfuscated, it might be easily captured by collecting all classes that have the same package name. However, as we have shown with app code in Chapter 4, capturing associated elements might not be straightforward. Consequently, one future direction could be to capture all associated elements of artifacts from obfuscated code. In many cases, it is desirable to find artifacts in a non-obfuscated form. As a result, future work needs to find a representation that captures the associated elements and develop a tool that can compare between the obfuscated and non-obfuscated forms of an artifact. For instance, *LibDetect* as well as *LibMapper* need non-obfuscated libraries to identify obfuscated ones in apps. We can check with *ObfusSpot* whether the code of a library is obfuscated. However, we cannot check whether an obfuscated and a non-obfuscated library represent the same code until we deobfuscated their names and checked with a previously found non-obfuscated library version. It would be useful to construct a representation that can identify obfuscated and non-obfuscated libraries as belonging to the same artifact.

As we have shown in Section 7.3, some libraries are only released in obfuscated form. To identify such libraries in the wild, we need to integrate them into our database. However, suppose we integrate obfuscated libraries into *LibMapper*. In that case, it does not help analysts to understand the code under analysis because it is hard for them to differentiate between obfuscated libraries and other obfuscated app code. As a consequence, future work could use name inference to assign names to the obfuscated code entities. Additionally, it could identify similar non-obfuscated libraries to transfer similar names to the obfuscated library, which would avoid assigning too generic names to the code entities.

Tool Support for Manual Analysis We manually analyzed many data sets for our experiments because many of the available analysis frameworks rely on complex languages or do not support the necessary capabilities such as navigation through code. Consequently, a future research direction could define a language that allows expressing complex static or dynamic analyses in a simple script-like fashion. Such a language could be similar to BOA [DNRN13]. However, instead of providing only data from source code and text files, it processes the sophisticated static and dynamic analyses of Android apps in the background and provides the results for further processing. A simple example of such a language could be XPath-like. In such a language, one could identify all methods with ten instructions using the expression `/Methods/Method[@instructionCount=10]`.

12.3. Handling Underlying Structural Changes

Obfuscators can change the code structures used by our approaches. Different ways to cope with this should be investigated in future directions. Suppose an obfuscator changes structures by manipulating some instructions or code entities such as control-flow flattening or virtualization of API methods. In that case, future work could use

automatic and safe refactoring to keep these structures stable. Furthermore, if obfuscators use translingual obfuscation [WWM⁺16] or virtualizes the code completely into another language, future work could try to hook the interpreter instructions to extract known structures as done by Coogan et al. [CLD11] for the C language.

The authors analyzed interpreters that virtualized software and were integrated into code to execute a guest language. Their tool removed conditions that would hinder the interpreter's execution, instrumented the interpreter's instructions, and executed the resulting interpreter with the guest code to get an execution trace of the host language. A similar approach could be used in future work to resolve translingual and virtualized obfuscation for Android apps. However, instead of extracting the execution traces, future approaches need to create stable code structures to identify libraries or repackaged apps.

12.4. Handling the Hiding of Functionality

If obfuscators encrypt or compress a part of the code and load it from an external resource, many of our approaches cannot identify libraries or the repackaged part of the code. Consequently, future work needs to extract the hidden code from the external location. Future approaches have to remove potential countermeasures that would hinder the extraction and hook the classloader for the extraction. While Dongwoo et al. [KKR15] already developed an approach that hooks classloaders, this approach could be evaded by state-of-the-art obfuscators.

For the evasion of the hooking approach, obfuscators may use the developer certificate [ZLQ⁺18] or other external resources from the app to prevent the execution of the extracted code. Future work could emulate the `android.jar` to counter such techniques. For the emulation of an entire device, analysts need to collect all information about a device. In contrast, the emulation of the `android.jar` would enable a future approach to execute only the necessary instructions without caring for the entire device. For instance, the emulated `android.jar` could use the same functionality as used in an app to extract the developer certificate. However, while such an approach could involve considerable engineering effort, it would benefit many future works.

12.5. Eliminating the Arms Race

The deobfuscation and obfuscation of code is an arms race against each other—both deobfuscation and obfuscation benefit attackers and analysts. Attackers use better obfuscation to hide malicious payloads and better deobfuscation to uncover developers' intellectual property. In contrast, analysts profit from better obfuscation to protect developers' intellectual property and better deobfuscation to uncover malicious payloads. Therefore, working on each of the techniques is handling a double edge sword.

If obfuscation distributes data across boundaries of methods, future work could collect such data using interprocedural analyses. Furthermore, if obfuscators remapped the names in an app randomly, future works would need to find a way to reverse it. When obfuscators use probing [XQE16] to identify vulnerabilities of classifiers that identify

12. Future Work

obfuscation, future approaches need to use features that are not affected by probing. What seems an endless arms race could be stopped by future obfuscation research that focuses on protecting intellectual property combined with analysability for malicious payloads. With such an obfuscation, the developers would separate themselves from malicious actors, and the apps of such actors would be easy to identify.

Contributed Implementations and Data

In this dissertation, we have developed several research prototypes and collected and analyzed multiple data sets. We provide these prototypes and data sets to support future research. We believe that this is good scientific practice and want to encourage other researchers to do the same.

All tools are developed in Scala, and as mentioned in the background, we used *Enjarify* [enj17] to transform APKs into JAR files and analyzed it using *OPAL* [EH14].

12.6. CodeMatch & LibDetect

We provide the implementation of *CodeMatch* and *LibDetect* in the following repository:

<https://github.com/stg-tud/CodeMatch-LibDetect>

Both tools can be started from the command line with the necessary parameters. To use *LibDetect*, one needs to store all known libraries in *LibDetect*'s database. To get started, we provided all libraries used in our experiments in a text file.

Additionally, if *CodeMatch* is used with *F2S2*, one needs to store all original apps into *F2S2*'s database.

12.7. StringHound, String Classifier & Method Classifier

In order to use *StringHound* along with the two classifiers, one needs to download their code from the following link.

<https://github.com/stg-tud/StringHound>

The classifiers are already trained on the data from our experiments, and all tools can be used from a single command line by using different flags. However, to install the *String Classifier*, one needs to merge the dictionary containing the 54 natural languages.

12.8. AppSeparator, LibMapper, & ObfusSpot

To use *AppSeparator*, *LibMapper* or *ObfusSpot*, one need to download their code from the following repository:

<https://github.com/stg-tud/ObfusSpot>

While the database of *LibMapper* needs to be filled with non-obfuscated libraries, the code of *AppSeparator* and *ObfusSpot* are directly usable.

Study Artifacts

In addition to the developed prototypes, we provide all necessary data to reproduce our experiments from the evaluation of *CodeMatch* and *LibDetect* at the following page: <http://www.st.informatik.tu-darmstadt.de/artifacts/codematch>

The hash sums of the 100,000 apps that we used for the evaluation of *StringHound*, *String Classifier*, *Method Classifier*, and *ObfusSpot* can be found in a text file of the following link.

<https://github.com/stg-tud/StringHound/blob/master/SHA256ForF-DroidApks.txt>

Bibliography

- [AARUJ86] V Aho Alfred, Sethi Ravi, and D Ullman Jeffrey. Compilers: Principles, techniques, and tools. *Reading: Addison Wesley Publishing Company*, 1986.
- [ABBS14] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 281–293. ACM, 2014.
- [ABBS15] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 38–49. ACM, 2015.
- [ABKLT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *In Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, MSR '16, pages 468–471, 2016.
- [Act20] Activity, Last accessed: 01/04/2020. <https://developer.android.com/reference/android/app/Activity>.
- [AL11] Sunita B Aher and LMRJ Lobo. Data mining in educational system using weka. In *In Proceedings of the 4th International Conference on Emerging Technology Trends (ICETT)*, volume 3, pages 20–25, 2011.
- [AL12] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *In Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, pages 688–712. Springer, 2012.
- [All19] Allatori java obfuscator, Last accessed: 05/15/2019. <http://www.allatori.com/>.
- [AMSS15] Alessandro Aldini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Detection of repackaged mobile applications through a collaborative approach. *Concurrency and Computation: Practice and Experience*, 27(11):2818–2838, 2015.
- [and17] Androguard, Last Accessed: 11/27/2017. <http://androguard.readthedocs.io/en/latest/>.

Bibliography

- [And19] Androidrank, Last accessed: 05/15/2019. <https://www.androidrank.org/>.
- [and20a] App manifest overview, Last accessed: 01/03/2020. <https://developer.android.com/guide/topics/manifest/manifest-intro>.
- [and20b] Android ABIs, Last accessed: 01/04/2020. <https://developer.android.com/ndk/guides/abis>.
- [and20c] Documentation for app developers, Last accessed: 10/12/2020. <https://developer.android.com/docs>.
- [And20d] Android Market: Now available for users, Last accessed: 07/14/2020. <https://android-developers.googleblog.com/2008/10/android-market-now-available-for-users.html>.
- [Ann20] Annual number of app downloads from the google play store worldwide from 2016 to 2019, Last accessed: 07/23/2020. <https://www.statista.com/statistics/734332/google-play-app-installs-per-year/>.
- [Anz17] Anzhi app marketplace, Last accessed: 01/11/2017. <http://www.anzhi.com/>.
- [apk20] Analyze your build with apk analyzer, Last accessed: 10/12/2020. <https://developer.android.com/studio/build/apk-analyzer>.
- [app17a] Apps builder, Last accessed: 01/11/2017. <http://www.apps-builder.com>.
- [App17b] App china app marketplace, Last accessed: 01/11/2017. <http://www.appchina.com/>.
- [App19] App Brain's Ad Networks, Last accessed: 05/15/2019. <https://www.appbrain.com/stats/libraries/ad-networks>.
- [ARF⁺14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.
- [ARSC16] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. Droidclone: Detecting android malware variants by exposing code clones. In *In Proceedings of the 6th International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 79–84. IEEE, 2016.

- [AZLY19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
- [Bas19a] The base16, base32, and base64 data encodings, Last accessed: 02/20/2019. <https://tools.ietf.org/html/rfc4648>.
- [Bas19b] A compact representation of ipv6 addresses, Last accessed: 02/20/2019. <https://tools.ietf.org/html/rfc1924>.
- [Bay91] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. 1763. *MD Computing: Computers in Medical Practice*, 8(3):157, 1991.
- [BBD16] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *In Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 356–367. ACM, 2016.
- [BG96] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43(1-50):1–2, 1996.
- [Boe99] Barry Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.
- [BP18] Luciano Bello and Marco Pistoia. Ares: Triggering payload of evasive android malware. In *In Proceedings of the IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, MOBILESoft’18, pages 2–12. IEEE, 2018.
- [Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [bro20] Broadcast Receiver, Last accessed: 01/04/2020. <https://developer.android.com/reference/android/content/BroadcastReceiver>.
- [BRTV16] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *In Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 343–355. ACM, 2016.
- [CADZ15] Jian Chen, Manar H Alalfi, Thomas R Dean, and Ying Zou. Detecting android malware using clone detection. *Journal of Computer Science and Technology*, 30(5):942–956, 2015.
- [CCS⁺15] Yang Cuixia, Zuo Chaoshun, Guo Shanqing, Hu Chengyu, and Cui Lizhen. Ui ripping in android: Reverse engineering of graphical user interfaces and its application. In *In Proceedings of the 2015 IEEE Conference on Collaboration and Internet Computing (CIC)*, pages 160–167. IEEE, 2015.

- [cer20] Sign your app, Last accessed: 01/04/2020. <https://developer.android.com/studio/publish/app-signing>.
- [CFL⁺17] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *In Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, NDSS'17, 2017.
- [CFPK20] Rhys Compton, Eibe Frank, Panagiotis Patros, and Abigail Koay. Embedding java classes with code2vec: Improvements from variable obfuscation. In *In Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. ACM, 2020.
- [CFR19] CFR, Last accessed: 05/15/2019. <http://www.benf.org/other/cfr/>.
- [CGC12] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security—ESORICS*, pages 37–54. Springer, 2012.
- [CGC13] Jonathan Crussell, Clint Gibler, and Hao Chen. Scalable semantics-based detection of similar android applications. In *In Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, volume 13. Citeseer, 2013.
- [Cha17] Marek Chalupa. Slicing of LLVM bitcode, Last accessed: 09/18/2017. https://is.muni.cz/th/396236/fi_m/thesis.pdf.
- [CLD11] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *In Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 275–284. The University of Arizona, 2011.
- [CLZ14] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *In Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 175–186. ACM, 2014.
- [con19a] ProGuard manual, Last accessed: 05/31/2019. <https://www.guardsquare.com/en/products/proguard/manual/usage>.
- [Con19b] Contagio Mobile Dump, Last accessed: 05/15/2019. <http://contagiomindump.blogspot.com/>.
- [con20] Content Provider, Last accessed: 01/04/2020. <https://developer.android.com/guide/topics/providers/content-providers>.
- [cry19] Practical Cryptography, Last accessed: 05/15/2019. <http://practicalcryptography.com/cryptanalysis/>.

- [Cry20] Cryptography, Last accessed: 02/12/2020. <https://developer.android.com/guide/topics/security/cryptography>.
- [CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.
- [CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 184–196. ACM, 1998.
- [CWC⁺16] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following devil’s footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *In Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, pages 357–376. IEEE, 2016.
- [CWL⁺15] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *In Proceedings of the 24th USENIX Security Symposium*, volume 15, pages 659–674, 2015.
- [CYL⁺17] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. Mass discovery of android traffic imprints through instantiated partial execution. In *In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS’17, pages 815–828. ACM, 2017.
- [dam20] Cybercrime Damages 6 Trillion Dollars By 2021, Last accessed: 07/14/2020. <https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/>.
- [Das19] DashO, Last accessed: 05/15/2019. <https://www.preemptive.com/>.
- [dec20] Decision Tree: How To Create A Perfect Decision Tree?, Last accessed: 01/04/2020. <https://www.edureka.co/blog/decision-trees/>.
- [dex19a] Dexprotector android obfuscator, Last accessed: 01/20/2019. <https://dexprotector.com>.
- [Dex19b] Dex2Jar Decrypt Strings, Last accessed: 05/15/2019. <https://sourceforge.net/p/dex2jar/wiki/DecryptStrings/>.
- [Dex19c] Dexguard android obfuscator, Last accessed: 02/20/2019. <https://www.guardsquare.com/en/dexguard>.
- [Dex20] Dex Oracle, Last accessed: 05/15/2020. <https://github.com/CalebFenton/dex-oracle>.

- [dic20] Dictionaries for Sublime Text, Last accessed: 01/04/2020. <https://github.com/titoBouzout/Dictionaries>.
- [DLR77] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.
- [DMY⁺16] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. Free for all! assessing user data exposure to advertising libraries on android. In *In Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, NDSS’16, 2016.
- [DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *In Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013.
- [Don18] Dong, Shuaike and Li, Menghao and Diao, Wenrui and Liu, Xiangyu and Liu, Jian and Li, Zhou and Xu, Fenghao and Chen, Kai and Wang, Xiaofeng and Zhang, Kehuan. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. pages 172–192, 2018.
- [dsa20] DIGITAL SIGNATURE STANDARD (DSS), Last accessed: 01/04/2020. <https://www.webcitation.org/66kuhq8Pb?url=http://www.itl.nist.gov/fipspubs/fip186.htm>.
- [EH14] Michael Eichberg and Ben Hermann. A software product line for static analyses: the opal framework. In *In Proceedings of the Third ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP)*, pages 1–6. ACM, 2014.
- [EHMG15] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 474–484, 2015.
- [EHPVS09] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. *IEEE Computer Society*, pages 71–80, 2009.
- [enj17] Enjarify, Last accessed: 01/11/2017. <https://github.com/google/enjarify>.
- [ErJ01] D Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, Network Working Group, 2001.
- [F-D19] F-Droid, Last accessed: 05/15/2019. <https://f-droid.org/>.

- [FBR⁺16] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *In Proceedings of the 37th Annual Symposium on Security and Privacy (SP)*, SP'16, pages 377–396. IEEE, 2016.
- [FGL⁺13] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. Androsimilar: Robust statistical feature signature for android malware detection. In *In Proceedings of the 6th International Conference on Security of Information and Networks (SINCONF)*, pages 152–159. ACM, 2013.
- [FLB⁺15] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, Manoj Singh Gaur, and Vijay Ganmoor. Androsimilar: Robust signature for detecting variants of android malware. *Journal of Information Security and Applications*, 22:66–80, 2015.
- [FR19] Johannes Feichtner and Christof Rabensteiner. Obfuscation-resilient code recognition in android apps (ares). In *In Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES)*, pages 1–10, 2019.
- [Fre17] Freeware lovers app marketplace, Last accessed: 01/11/2017. <http://www.freewarelovers.com/>.
- [GAE⁺17] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: Obfuscation won't conceal your repackaged app. In *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 638–648. ACM, 2017.
- [GHLZ16] Quanlong Guan, Heqing Huang, Weiqi Luo, and Sencun Zhu. Semantics-based repackaging detection for mobile apps. In *In Proceedings of the 8th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 89–105. Springer, 2016.
- [git20] GitHub, Last accessed: 07/15/2020. <https://github.com/>.
- [GKS⁺15] Hugo Gonzalez, Andi A Kadir, Natalia Stakhanova, Abdullah J Alzahrani, and Ali A Ghorbani. Exploring reverse engineering symptoms in android apps. In *In Proceedings of the 8th European Workshop on System Security (EuroSec)*, page 7. ACM, 2015.
- [GLZ16] Olga Gadyatskaya, Andra-Lidia Lezza, and Yury Zhauniarovich. Evaluation of resource-based app repackaging detection in android. In *In Proceedings of the 21st Nordic Conference on Secure IT Systems (NordSEC)*, pages 135–151. Springer, 2016.

Bibliography

- [GM17] Saad Mohamed Ali Mohamed Gadal and Rania A Mokhtar. Anomaly detection approach using hbrid algorithm of data mining technique. In *In Proceedings of the 2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*, pages 1–6. IEEE, 2017.
- [GMB⁺20] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*. ACM, 2020. accepted for publication.
- [gro20] Growth of available mobile apps at Google Play worldwide from 2nd quarter 2015 to 1st quarter 2020, Last accessed: 07/13/2020.
- [GS10] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 147–156. ACM, 2010.
- [GSC⁺13] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *In Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 431–444. ACM, 2013.
- [GSWH15] Leonid Glanz, Sebastian Schmidt, Sebastian Wollny, and Ben Hermann. A vulnerability’s lifetime: Enhancing version information in cve databases. In *In Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business (I-KNOW)*, page 28. ACM, 2015.
- [gui17] Fast library identification and recognition technology (1997), Last accessed: 01/11/2017. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.
- [GZZ⁺12] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 281–294. ACM, 2012.
- [HAB⁺16] Médéric Hurier, Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *In Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, DIMVA’16, pages 142–162. Springer, 2016.

- [HFH⁺09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining osftware: An update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.
- [HHW⁺12] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *In Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 62–81. Springer, 2012.
- [HiA17] Hiapk app marketplace, Last accessed: 01/11/2017. <http://www.hiapk.com/>.
- [HLT18] Hongmu Han, Ruixuan Li, and Junwei Tang. Identify and inspect libraries in android applications. *Wireless Personal Communications*, 103(1):491–503, 2018.
- [HØ09] Einar W Høst and Bjarte M Østvold. Debugging method names. In *In Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 294–317. Springer, 2009.
- [HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2009.
- [HUHS13] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, SAC ’13, pages 1844–1851, New York, NY, USA, 2013. ACM.
- [ICU19] ICU Tokenizer, Last accessed: 05/15/2019. <https://www.elastic.co/guide/en/elasticsearch/plugins/current/analysis-icu-tokenizer.html>.
- [int20a] Intent Filter, Last accessed: 01/04/2020. <https://developer.android.com/guide/topics/manifest/intent-filter-element>.
- [int20b] IntelliJ IDE, Last accessed: 02/12/2020. <https://www.jetbrains.com/de-de/idea/>.
- [IWAM16] Yuta Ishii, Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. Clone or relative?: Understanding the origins of similar android apps. In *In Proceedings of the 2016 ACM International Workshop on Security And Privacy Analytics (IWSPA)*, pages 25–32. ACM, 2016.
- [IWAM17] Yuta Ishii, Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. Appraiser: A large scale analysis of android clone apps. *IEICE TRANSACTIONS on Information and Systems*, 100(8):1703–1713, 2017.

Bibliography

- [Jaf17] Alan Jaffe. Suggesting meaningful variable names for decompiled code: A machine translation approach. In *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 1050–1052. ACM, 2017.
- [JAR20] jarg - java archive grinder, Last accessed: 06/15/2020. <http://jarg.sourceforge.net/>.
- [Jav17] Java DeObfuscator, Last accessed: 11/21/2017. <https://github.com/java-deobfuscator/deobfuscator>.
- [Jav20a] Java bytecode analysis/deobfuscation tool, Last accessed: 05/15/2020. <https://github.com/contra/JMD>.
- [Jav20b] Java Decompiler, Last accessed: 01/04/2020. <http://jd.benow.ca/>.
- [JL95] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *In Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.
- [jni20] Java Native Interface Specification, Last accessed: 01/04/2020. <https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>.
- [Kah96] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Simon and Schuster, 1996.
- [KCS16] Richard Killam, Paul Cook, and Natalia Stakhanova. Android malware classification through analysis of string literals. *Text Analytics for Cybersecurity and Online Safety (TA-COS)*, 2016.
- [KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.
- [KGG16] Daeyoung Kim, Amruta Gokhale, Vinod Ganapathy, and Abhinav Srivastava. Detecting plagiarized mobile apps using api birthmarks. *Automated Software Engineering*, 23(4):591–618, 2016.
- [KKR15] Dongwoo Kim, Jin Kwak, and Jaecheol Ryou. Dwroiddump: Executable code extraction from android applications for malware analysis. *International Journal of Distributed Sensor Networks*, 11(9), 2015.
- [KLCP19] Byoungchul Kim, Kyeonghwan Lim, Seong-Je Cho, and Minkyu Park. Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app’s manifest file. *Access*, 7:72182–72196, 2019.

- [kme20] sklearn.cluster.KMeans, Last accessed: 01/04/2020. <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>.
- [KNGS18] Ratinder Kaur, Ye Ning, Hugo Gonzalez, and Natalia Stakhanova. Unmasking android obfuscation tools using spatial analysis. In *In Proceedings of the 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2018.
- [Koe09] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2009.
- [Kor06] Jesse Kornblum. Identifying almost identical ifles using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.
- [Kun14] Ludmila I Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley & Sons, 2014.
- [LBK19] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *Transactions on Software Engineering*, 2019.
- [LCVH92] Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(1):191–201, 1992.
- [lea17] Leapdroid, Last accessed: 01/11/2017. <http://www.leapdroid.com>.
- [LH07] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *Security & Privacy*, 5(2):40–45, 2007.
- [LKL⁺16] Li Li, Jacques Klein, Yves Le Traon, et al. An investigation into the use of common libraries in android apps. In *In Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414. IEEE, 2016.
- [LLB⁺16] Li Li, Daoyuan Li, Tegawendé François D Assise Bissyande, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting malicious code from piggybacked android apps. Technical report, SnT, 2016.
- [LLB⁺17] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *Transactions on Information Forensics & Security (TIFS)*, 12(6):1269–1284, 2017.
- [LLCT13] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39:340–350, 2013.

- [LLJG15] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *In Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 89–103, 2015.
- [LLTX17] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. Reflection analysis for java: Uncovering more reflective targets precisely. In *In Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 12–23. IEEE, 2017.
- [LLYZ16] Fang Lyu, Yapin Lin, Junfeng Yang, and Junhai Zhou. Suidroid: An efficient hardening-resilient approach to android app clone detection. In *In Proceedings of the 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering, 14th IEEE International Symposium on Parallel and Distributed Processing with Applications (TrustCom/BigDataSE/ISPA)*, pages 511–518. IEEE, 2016.
- [LMP01] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *In Proceedings of the 18th International Conference on Machine Learning (ICML)*, pages 282–289. ACM, 2001.
- [LS11] Bart Lamiroy and Tao Sun. Computing precision and recall with missing or uncertain ground truth. In *In Proceedings of the 9th International Workshop on Graphics Recognition (GREC)*, pages 149–162. Springer, 2011.
- [LVHBCP14] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 242–251, 2014.
- [LVHP16] Mario Linares-Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. On automatically detecting similar android apps. In *In Proceedings of the 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.
- [LWW⁺17] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE Press, 2017.
- [Mav17] Maven central, Last accessed: 01/11/2017. <http://search.maven.org/>.
- [MD01] Eric Mjolsness and Dennis DeCoste. Machine learning for science: State of the art and future prospects. *Science*, 293(5537):2051–2055, 2001.

- [Mer87] Ralph C Merkle. A digital signature based on a conventional encryption function. In *In Proceedings of the 4th Conference on the Theory and Application of Cryptographic Techniques (Eurocrypt)*, pages 369–378. Springer, 1987.
- [Mil95] George A Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.
- [Mir18] Mirzaei, O and de Fuentes, JM and Tapiador, J and Gonzalez-Manzano, L. AndrODet: An Adaptive Android Obfuscation Detector. *Future Generation Computer Systems*, 2018.
- [MJ15] Ms Urvashi Modi and Anurag Jain. A survey of ids classification using kdd cup 99 dataset in weka. *International Journal of Scientific & Engineering Research*, 6(11):947–954, 2015.
- [MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *In Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC)*, pages 421–430. IEEE, 2007.
- [MKS19] Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. Robust detection of obfuscated strings in android apps. In *In Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 25–35. ACM, 2019.
- [MMR19] Michael Meli, Matthew R McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. In *In Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.
- [MS04] Leann Myers and Maria J Sirois. Spearman correlation coefficients, differences between. *Encyclopedia of Statistical Sciences*, 12, 2004.
- [mus17] Must-have libraries, Last accessed: 02/24/2017. https://github.com/codepath/android_guides/wiki/Must-Have-Libraries.
- [MW17] Luis Menezes and Roland Wismüller. Detecting information leaks in android applications using a hybrid approach with program slicing, instrumentation and tagging. In *In Proceedings of the 51th International Carnahan Conference on Security Technology (ICCST)*, pages 1–6. IEEE, 2017.
- [MWGC16] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, pages 653–656. ACM, 2016.

- [MZW⁺16] Jiang Ming, Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *Transactions on Reliability*, 65(4):1647–1664, 2016.
- [nat20] Getting Started with the NDK, Last accessed: 01/04/2020. <https://developer.android.com/ndk/guides>.
- [NCC14] Arun Narayanan, Lihui Chen, and Chee Keong Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *In Proceedings of the IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE, 2014.
- [num20] Number of available applications in the Google Play Store from December 2009 to June 2020 , Last accessed: 07/07/2020. <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>.
- [NYW⁺18] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, NDSS’18, 2018.
- [ODS⁺18] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. The rise of the citizen developer: Assessing the security impact of online app generators. In *In Proceedings of the 39th Annual Symposium on Security and Privacy (SP)*, pages 634–647. IEEE, 2018.
- [OMA⁺19] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *Transactions on Privacy and Security (TOPS)*, 22(2):1–34, 2019.
- [ora19] Oracle naming conventions, Last accessed: 04/26/2019. <https://www.oracle.com/technetwork/java/codeconventions-135099.html>.
- [pai20] Distribution of free and paid Android apps in the Google Play Store as of June 2020, Last accessed: 07/07/2020. <https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/>.
- [per20] Permissions overview, Last accessed: 01/04/2020. <https://developer.android.com/guide/topics/permissions/overview>.

- [PHD11] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, pages 73–82. ACM, 2011.
- [pla17a] Google play store apps in the playdrone archive, Last accessed: 01/11/2017. <https://archive.org/details/playdrone-apks>.
- [pla17b] Playdrone archive snapshot 10/31/2014, Last accessed: 01/11/2017. <http://archive.org/download/playdrone-snapshots/2014-10-31.json>.
- [PNNRZ12] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. *Engineering Secure Software and Systems*, pages 106–120, 2012.
- [Pri77] Wyn L. Price. A controlled random search procedure for global optimisation. *The Computer Journal*, 20(4):367–370, 1977.
- [Pro17] Proguard, Last accessed: 01/11/2017. <http://proguard.sourceforge.net/>.
- [Pro19] Protecting users with tls by default in android p, Last accessed: 11/22/2019. <https://android-developers.googleblog.com/2018/04/protecting-users-with-tls-by-default-in.html>.
- [pro20a] Android developers <http://developer.android.com/tools/help/proguard.html>, Last accessed: 02/21/2020. <http://developer.android.com/tools/help/proguard.html>.
- [Pro20b] Proguard provides minimal obfuscation. dexguard applies multiple layers of encryption and obfuscation., Last accessed: 02/18/2020. <https://www.guardsquare.com/en/blog/dexguard-vs-proguard>.
- [PWD⁺17] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In *In Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, NDSS’17, 2017.
- [PYC⁺19] Minjae Park, Geunha You, Seong-je Cho, Minkyu Park, and Sangchul Han. A framework for identifying obfuscation techniques applied to android apps using machine learning. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 10(4):22–30, 2019.
- [QPX⁺16] Zhongyuan QIN, Wanpeng PAN, Ying XU, Kerong FENG, and Zhongyun YANG. An efficient scheme of detecting repackaged android applications. *ZTE Communications*, 3:012, 2016.

- [Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
- [Qui96] J Ross Quinlan. Learning decision tree classifiers. *Computing Surveys (CSUR)*, 28(1):71–72, 1996.
- [RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *In Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, volume 14, page 1125. Citeseer, 2014.
- [RAK⁺15] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden, and Philipp Richter. Droidsearch: A tool for scaling android app triage to real-world app stores. In *In Proceedings of the 2015 Science and Information Conference (SAI)*, pages 247–256. IEEE, 2015.
- [RAMB16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *In Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.
- [rep20] REPTree, Last accessed: 01/04/2020. <https://www.dbs.ifi.lmu.de/~zimek/diplomathesis/implementations/EHNDs/doc/weka/classifiers/trees/REPTree.html>.
- [res20] App resources overview, Last accessed: 01/04/2020. <https://developer.android.com/guide/topics/resources/providing-resources>.
- [rev17] Codematch artifacts, Last accessed: 06/30/2017. <http://www.st.informatik.tu-darmstadt.de/artifacts/codematch/>.
- [RKE⁺19] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA. ACM, 2019.
- [RNV⁺18] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, and Christian Kreibich Phillipa Gill. Apps, trackers, privacy, and regulators. In *In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, volume 2018, 2018.
- [RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

- [RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.
- [RVK15] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 111–124. ACM, 2015.
- [Sch07] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2007.
- [ser20] Services Overview, Last accessed: 01/04/2020. <https://developer.android.com/guide/components/services>.
- [SGC⁺12] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *In Proceedings of the First Workshop on Mobile Security Technologies (MoST)*, volume 10 of *MoST'12*, 2012.
- [Shi20] Shield4j - a java class and android apk obfuscator, encrypter, shrinker and merger, Last accessed: 02/12/2020. <https://dzone.com/articles/shield4j-java-class-and>.
- [sim20] Simplify, Last accessed: 05/15/2020. <https://github.com/CalebFenton/simplify>.
- [SKK⁺16] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *Computing Surveys (CSUR)*, 49(1):4, 2016.
- [SKS16] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *In Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, NDSS'16, 2016.
- [SLGL09] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *In Proceedings of the 30th IEEE Symposium on Security and Privacy (SP)*, pages 94–109. IEEE, 2009.
- [SLL15] Mingshen Sun, Mengmeng Li, and John Lui. Droideagle: Seamless detection of visually similar android apps. In *In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, page 9. ACM, 2015.
- [SLQ⁺14] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *In Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, pages 56–65. ACM, 2014.

- [STA⁺16] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Anamalai Narayanan, and Lipo Wang. Libsift: Automated detection of third-party libraries in android applications. In *In Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 41–48. IEEE, 2016.
- [Sta20] Statistical data analysis in python, Last accessed: 06/10/2020. <https://github.com/fonnesbeck/statistical-analysis-python-tutorial/tree/master/data/names>.
- [STAW15] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. Detecting clones in android applications through analyzing user interfaces. In *In Proceedings of the 23rd International Conference on Program Comprehension (ICPC)*, pages 163–173. IEEE Press, 2015.
- [STDA⁺17] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *In Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 309–320. ACM, 2017.
- [str19] Stringer Java Obfuscator, Last accessed: 05/15/2019. <https://jfxstore.com/>.
- [SZX⁺14] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Detecting code reuse in android applications using component-based control flow graph. In *In Proceedings of the 29th IFIP International Information Security Conference (SEC)*, pages 142–155. Springer, 2014.
- [TLS17] Dennis Titze, Michael Lux, and Julian Schuette. Ordol: Obfuscation-resilient detection of libraries in android applications. In *In Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 11th IEEE International Conference on Big Data Science and Engineering, 14th IEEE International Conference on Embedded Software and Systems (Trustcom/BigDataSE/ICESS)*, pages 618–625. IEEE, 2017.
- [UPSB⁺11] Xabier Ugarte-Pedrero, Igor Santos, Pablo G Bringas, Mikel Gastesi, and José Miguel Esparza. Semi-supervised learning for packed executable detection. In *In Proceedings of the 5th International Conference on Network and System Security (NSS)*, pages 342–346. IEEE, 2011.
- [URL20] Urlencoder, Lastaccessed: 02/12/2020. <https://developer.android.com/reference/java/net/URLEncoder>.
- [VC14] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection (asia-ccs). In *In Proceedings of the 9th ACM Symposium*

- on Information, Computer and Communications Security, pages 447–458. ACM, 2014.
- [VCD17] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 683–693. ACM, 2017.
- [vir19] VirusTotal, Last accessed: 05/15/2019. <https://www.virustotal.com/>.
- [Wan05] Lipo Wang. *Support Vector Machines: Theory and Applications*, volume 177. Springer Science & Business Media, 2005.
- [Wei81] Mark Weiser. Program Slicing. In *In Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE Press, 1981.
- [WF02] Ian H Witten and Eibe Frank. Data mining: Practical machine learning tools and techniques with java implementations. *Sigmod Record*, 31(1):76–77, 2002.
- [WGMC15] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *In Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 71–82. ACM, 2015.
- [WHA⁺18] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, ACSAC’18, pages 222–235. ACM, 2018.
- [WL16] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *In Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, volume 16 of *NDSS’16*, pages 21–24, 2016.
- [WL18] Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in android with tiro. In *In Proceedings of the 27th USENIX Security Symposium*, USENIX Security’18, pages 1247–1262, 2018.
- [WLG⁺17] Chundong Wang, Zhiyuan Li, Liangyi Gong, Xiuliang Mo, Hong Yang, and Yi Zhao. An android malicious code detection method based on improved dca algorithm. *Entropy*, 19(2):65, 2017.
- [WR17] Yan Wang and Atanas Rountev. Who changed you?: Obfuscator identification for android. In *In Proceedings of the IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 154–164, 2017.

- [WSY13] Christian Winter, Markus Schneider, and York Yannikos. F2s2: Fast forensic similarity search through indexing piecewise hash signatures. *Digital Investigation*, 10(4):361–371, 2013.
- [WWM⁺16] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. Translingual obfuscation. In *In Proceedings of the First IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 128–144. IEEE, 2016.
- [WWZR18] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-resilient library detection for android. In *In Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 13–23. IEEE, 2018.
- [WZSL15] Xueping Wu, Dafang Zhang, Xin Su, and WenWei Li. Detect repackaged android application based on http traffic similarity. *Security and Communication Networks*, 8(13):2257–2266, 2015.
- [XQE16] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *In Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, pages 21–24, 2016.
- [YDGPS16] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *In Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.
- [YFM⁺17] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. Repdroid: An automated tool for android application repackaging detection. In *In Proceedings of the 25th International Conference on Program Comprehension (ICPC)*, pages 132–142. IEEE Press, 2017.
- [yGu20] yguard - java bytecode obfuscator and shrinker, Last accessed: 06/15/2020. <https://www.yworks.com/products/yguard>.
- [ZBK19] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. Libid: Reliable identification of obfuscated third-party android libraries. In *In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 55–65. ACM, 2019.
- [ZDZ⁺18] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *In Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.

- [Zel19] Zelix KlassMaster, Last accessed: 05/15/2019. <http://www.zelix.com/>.
- [ZGC⁺14] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: Fast detection of repackaged applications. In *In Proceedings of the 28th Annual International Working Conference on Data and Applications Security and Privacy (DBSec)*, pages 130–145. Springer, 2014.
- [ZGW⁺16] Jingqiu Zheng, Keyu Gong, Songhe Wang, Yifei Wang, and Min Lei. Repackaged apps detection based on similarity evaluation. In *In Proceedings of the 8th International Conference on Wireless Communications & Signal Processing (WCSP)*, pages 1–5. IEEE, 2016.
- [ZHZ⁺14] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection (wisec). In *In Proceedings of the 7th ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec)*, pages 25–36. ACM, 2014.
- [ZLQ⁺18] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. Resilient decentralized android application repackaging detection using logic bombs. In *In Proceedings of the 16th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 50–61, 2018.
- [ZLZ19] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *In Proceedings of the 40th Annual Symposium on Security and Privacy (SP)*, SP’19, 2019.
- [ZSL13] Min Zheng, Mingshen Sun, and John CS Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *In Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 163–171. IEEE, 2013.
- [ZWWJ15] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, page 23. ACM, 2015.
- [ZWZJ14] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *In Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 199–210. ACM, 2014.

Bibliography

- [ZZG⁺13] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *In Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 185–196. ACM, 2013.
- [ZZJN12] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *In Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 317–326. ACM, 2012.
- [ZZPZ19] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Li Zhiqiang. Geo-locating drivers: A study of sensitive data leakage in ride-hailing services. In *In Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, NDSS’19, 2019.

Appendix

A. BOA Script for ProGuard Mapping Files

We used the following BOA script to search all GitHub repositories from 2019 for mapping files. With these files, *ProGuard* shows developers how it manipulated the code entities and under which entry a name can be found in an obfuscated app. Developers often use these files to understand the names from crash reports received from users. By default, the Android build system uses the folder `proguard` to store these files, and they are often named `mapping.txt`. Therefore, we check for similar patterns of these two default settings and transform the location of such files into a link, which makes the file downloadable.

```
1
2 p: Project = input;
3 counts: output set of string;
4 c:CodeRepository;
5 r:Revision;
6 visit(p,visitor {
7     before codeR: CodeRepository -> c=codeR;
8     before rev: Revision -> r=rev;
9     before f:ChangedFile -> {
10         if(match(".*proguard.*map.*$",f.name))
11             counts << format("%s/blob/%s/%s",c.url,r.id,f.name);
12     }
13 }
14 );
```

Listing 12.1: BOA script for name obfuscation mapping files

B. Bytecode Instructions to SPR

We used the following mappings, from bytecode (BC) instructions to *Structure-Preserving Representation* (SPR), as representations for our tools.

Table 12.1.: Part I of the mapping from Bytecode to SPR

BC	SPR	BC	SPR	BC	SPR
putstatic	put	i2d	to	lload_2	load
putfield	put	i2f	to	lload_3	load
getfield	get	i2l	to	aaload	arrayload
getstatic	get	i2s	to	baload	arrayload
ldc	ldc	l2d	to	caload	arrayload
ldc2_w	ldc	l2f	to	daload	arrayload
ldc_w	ldc	l2i	to	faload	arrayload
invokeinterface	invoke	aload	load	iaload	arrayload
invokespecial	invoke	aload_0	load	laload	arrayload
invokestatic	invoke	aload_1	load	saload	arrayload
invokevirtual	invoke	aload_2	load	astore	store
new	new	aload_3	load	astore_0	store
anewarray	newarray	dload	load	astore_1	store
multianewarray	newarray	dload_0	load	astore_2	store
newarray	newarray	dload_1	load	astore_3	store
arraylength	arraylength	dload_2	load	dstore	store
athrow	athrow	dload_3	load	dstore_0	store
bipush	push	fload	load	dstore_1	store
sipush	push	fload_0	load	dstore_2	store
nop	nop	fload_1	load	dstore_3	store
checkcast	check	fload_2	load	fstore	store
instanceof	check	fload_3	load	fstore_0	store
d2f	to	iload	load	fstore_1	store
d2i	to	iload_0	load	fstore_2	store
d2l	to	iload_1	load	fstore_3	store
f2d	to	iload_2	load	istore	store
f2i	to	iload_3	load	istore_0	store
f2l	to	lload	load	istore_1	store
i2b	to	lload_0	load	istore_2	store
i2c	to	lload_1	load	istore_3	store

Table 12.2.: Part II of the mapping from Bytecode to SPR

BC	SPR	BC	SPR	BC	SPR
lstore	store	ddiv	div	ifgt	if
lstore_0	store	fdiv	div	ifle	if
lstore_1	store	idiv	div	iflt	if
lstore_2	store	ldiv	div	ifne	if
lstore_3	store	dmul	mul	ifnonnull	if
aastore	arraystore	fmul	mul	ifnull	if
bastore	arraystore	imul	mul	jsr	if
castore	arraystore	lmul	mul	jsr_w	if
dastore	arraystore	ishl	mul	goto	if
fastore	arraystore	ishr	mul	goto_w	if
iastore	arraystore	iushr	mul	dup	dup
lastore	arraystore	lshl	mul	dup_x1	dup
sastore	arraystore	lshr	mul	dup_x2	dup
areturn	return	lushr	mul	dup2	dup
dreturn	return	dneg	neg	dup2_x1	dup
freturn	return	fneg	neg	dup2_x2	dup
ireturn	return	ineg	neg	iand	and
lreturn	return	lneg	neg	land	and
return	return	drem	rem	ior	or
ret	return	frem	rem	lor	or
aconst_null	const	irem	rem	ixor	xor
dconst_0	const	lrem	rem	lxor	xor
dconst_1	const	dsub	sub	lookupswitch	switch
fconst_0	const	fsub	sub	tableswitch	switch
fconst_1	const	isub	sub	monitorenter	monitor
fconst_2	const	lsub	sub	monitorexit	monitor
iconst_m1	const	dcmpg	if	pop	pop
iconst_0	const	dcmpl	if	pop2	pop
iconst_1	const	fcmpg	if	swap	swap
iconst_2	const	fcmpl	if	wide	wide
iconst_3	const	lcmp	if		
iconst_4	const	if_acmpeq	if		
iconst_5	const	if_acmpne	if		
lconst_0	const	if_icmpeq	if		
lconst_1	const	if_icmpge	if		
dadd	add	if_icmpgt	if		
fadd	add	if_icmple	if		
iadd	add	if_icmplt	if		
ladd	add	if_icmpne	if		
iinc	add	ifge	if		